TRUSTONIC

<t-base
Provisioning API

PREFACE

This specification is the confidential and proprietary information of Trustonic ("Confidential Information"). This specification is protected by copyright and the information described therein may be protected by one or more EC patents, foreign patents, or pending applications. No part of the Specification may be reproduced or divulged in any form by any means without the prior written authorization of Trustonic. Any use of the Specification and the information described is forbidden (including, but not limited to, implementation, whether partial or total, modification, and any form of testing or derivative work) unless written authorization or appropriate license rights are previously granted by Trustonic.

TRUSTONIC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF SOFTWARE DEVELOPED FROM THIS SPECIFICATION, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. TRUSTONIC SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SPECIFICATION OR ITS DERIVATIVES.

VERSION HISTORY

Version	Date	Status	Modification
1.5	25 Feb 2013	Issued	Adapted from earlier 1.4 document to Trustonic template

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 LEVEL 1

Acronyms

ADB	**A**ndroid **D**ebug **B**ridge
AES	**A**dvanced **E**ncryption **S**tandard
API	**A**pplication **P**rogramming **I**nterface
CBC	**C**ipher **B**lock **C**haining
DLL	**D**ynamic **L**ink **L**ibrary (MS Windows)
HSM	**H**ardware **S**ecurity **M**odule
KPH	**K**ey **P**rovisioning **H**ost
NVM	**N**on-**V**olatile **M**emory
NWd	**N**ormal **W**orl**d**
PGP	**P**retty **G**ood **P**rivacy
PRNG	**P**seudo **R**andom **N**umber **G**enerator
PSE	**P**ersonal **S**ecurity **E**nvironment
RFU	**R**eserved for **F**uture **U**se
RNG	**R**andom **N**umber **G**enerator
SO	**S**ecure **O**bject
SoC	**S**ystem **o**n a **C**hip
SUID	**S**oC **U**nique **ID**entifier
SWd	**S**ecure **W**orl**d**
Soft-PSE	**Soft**ware-**PSE** (e.g. PKCS#12/PFX container)
TEE	**T**rusted **E**xecution **E**nvironment
TLS	**T**ransport **L**ayer **S**ecurity
TRNG	**T**rue **R**andom **N**umber **G**enerator
USB	**U**niversal **S**erial **B**us

WSM	**W**orld **S**hared **M**emory

1 INTRODUCTION

1.1 PURPOSE OF THIS DOCUMENT

The document in-hand defines the **A**pplication **P**rogramming **I**nterface (API) of the MobiCore® Provisioning System. The key provisioning – aka "*device binding*" –, the associated security architecture, and the key management are explained in **Error! Reference source not found.**.

1.2 ROLES AND ENTITIES

1.2.1 Role "OEM"

The OEM is the device manufacturer and operates one or more production facilities.

1.2.2 Role "Vendor"

Giesecke & Devrient is the vendor of the TEE (MobiCore®) and provides the Provisioning API to OEMs.

1.2.3 Entity "Device"

The Device is the smartphone, tablet PC, netbook, etc. MobiCore® is applied to. A rich OS (e.g. Android) is executed in the Normal World (NWd), whereas MobiCore® runs in the Secure World (SWd).

1.2.4 Entity "Production Station"

The Production Station connects to one or more devices, performs the initial configuration of the device (e.g. IMEI, S/N, etc.), and stores the final OS image in the flash of the device (if multiple images are required by the different production process stages).

1.2.5 Entity "Production Network"

The Production Network interconnects the Production Stations and any additional backend systems the OEM might have added to the production (e.g. database).

1.2.6 Entity "Key Provisioning Host" (formerly "HSM Station")

The ***K**ey **P**rovisioning **H**ost* (KPH) delivers cryptographically strong random numbers and performs symmetric and asymmetric enciphering or signing, respectively. Moreover, a smartcard holding an AES-256bit key is used to unlock the KPH, i.e. RSA private keys are never stored in plain text.

The Key Provisioning Host is not part of the prototype (please refer to subsection 1.4.1 on page 13). The KPH prototype is a pure software implementation that is part of the MS Windows DLL (please refer to the next section 1.4).

Please refer to the *MobiCore® Key Provisioning Host (KPH) Operating Manual* for more information about the (hardware) KPH (refer to **Error! Reference source not found.**).

1.3 KEY PROVISIONING "*IN A NUTSHELL*"

*Please also refer to the MobiCore® Security Architecture (**Error! Reference source not found.**).*

MobiCore® is mainly based on symmetric cryptography regarding the MobiCore® Content Management System (CMS).

Cryptographic operations include (but are not limited to):

- Symmetric encryption: AES (FIPS 197); key size: 256 bits
- Message digesting: SHA-256 (FIPS 180-2); MD size: 256 bits
- Integrity protection: HMAC (FIPS 198) with SHA-256 (MAC size: 256 bits)

A MobiCore®-enabled Mobile Device performs content management communication with the G&D Backend Systems[1]. **Mutual** authentications are performed between the Mobile Device and the G&D Backend Systems in the field authenticating both communication partners.

The mutual authentication mechanism requires an **initial** key to be known by both systems (Mobile Device and Backend System). This initial key is also known as the "*trust anchor*".

Each MobiCore®-enabled Mobile Device is equipped with two items:

1. A 256bit master AES key internally stored in the SoC and **not exportable** (this key is only known by and can only be used internally by the CPU);
2. A 128bit globally unique identifier called the SUID (**S**oC **U**nique **ID**).

Please disregard the first item for now. You can read more about the master key in **Error! Reference source not found.**. The second item (SUID) is very important to **uniquely identify** a specific Mobile Device in the Backend System. Both items are part of the SoC.

1 The Backend systems are not operated by G&D only but also by 3rd parties.

The key provisioning generates a new 256bit AES key (in the KPH) and binds it to the SUID of the SoC – for this reason, the process is also called "*device binding*".

The 256bit AES key is transferred to the Mobile Device, which wraps this key in a so called "Secure Object" (SO) that is stored in the device. Furthermore, the KPH wraps the 256bit AES key in an RSA-envelope (RSA-encrypted and –signed). The latter one is called "*the receipt*" (technically: SD.Receipt) and has to be transferred by the OEM to the G&D Backend System.

When the Mobile Device connects to the G&D Backend System later on (in the field), it reads this 256bit AES key from the Secure Object (unwrapping). The G&D Backend System gets the key from its database. The key is uniquely identified by the SUID of the SoC, which is the primary key in the G&D backend database.

This 256bit AES key has to be generated in the OEM production because the OEM environment is trustworthy. The Mobile Device proves by the knowledge of this key that it is a **genuine device**.

The key is called "*K.SoC.Auth*", which stands for "*SoC Authentication Key*".

1.4 MOBICORE® PROVISIONING API OVERVIEW

The MobiCore® Provisioning API is split into two parts but implemented as one homogenous API (identical API functions for all supported platforms):

MS Windows (32bit/64bit) / Linux (32bit/64bit) API provided as a dynamic link library[2] (DLL) to be integrated by the OEM in the production software (platform: x86 or x86-64, respectively) – this is the KPH Connector in

1. Figure 1 on page 14;
2. Android API (32bit) provided as a shared object (so) to be integrated by the OEM in an executable unit of the flash image of the device (platform: ARM core).

Important note:

If Android is **not** available during the production of the devices, then the OEM is obliged to provide detailed specifications of the operating system or boot loader environment that is deployed to the devices for their production.

The above mentioned software libraries implement a cryptographic protocol that consists of several data packets, which are exchanged between the device and the Production Station.

Figure 1 illustrates the provisioning system as a whole. The light blue components in

[2] Shared Object in case of Linux

Figure 1 will be provided by G&D, the gray parts have to be added or provided by the OEM, respectively.

1.4.1 The KPH Connector

Please note that

Figure 1 shows the productive version of the provisioning system. The KPH Connector represents the MS Windows DLL.

Two different versions of the KPH Connector exist:

a. The **prototype version** of the KPH Connector encapsulates both the API and the Key Provisioning Host in one single DLL or shared object, respectively. The cryptographic functionality is implemented in software.

The prototype version **cannot** be used to produce end-customer devices because the keying material of the DLL is neither recognized nor accepted by the G&D Backend Systems. This version can be deployed in **test production environments** only.

b. The **production version** of the KPH Connector exposes the API only. The cryptographic operations are performed by the Key Provisioning Host (a hardware appliance) over a TCP/IP connection.

Please note:

The integration of the APIs has to be performed by the OEM only **once**. The API does not change from the prototype to the production version of the provisioning system. Only a software update of the Windows DLL (or Linux SO) is required.

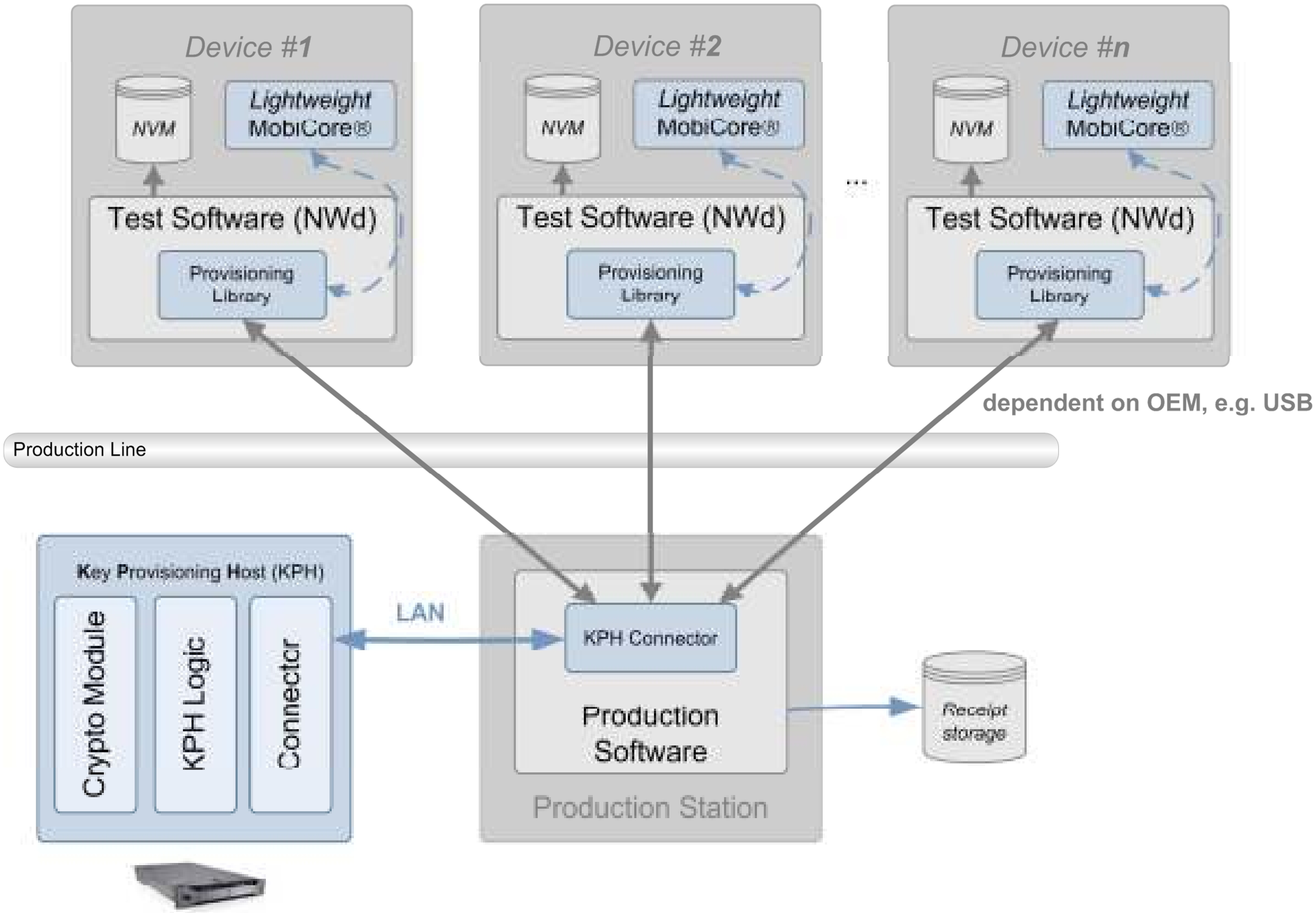

Figure 1: The provisioning system fully integrated in the production line.

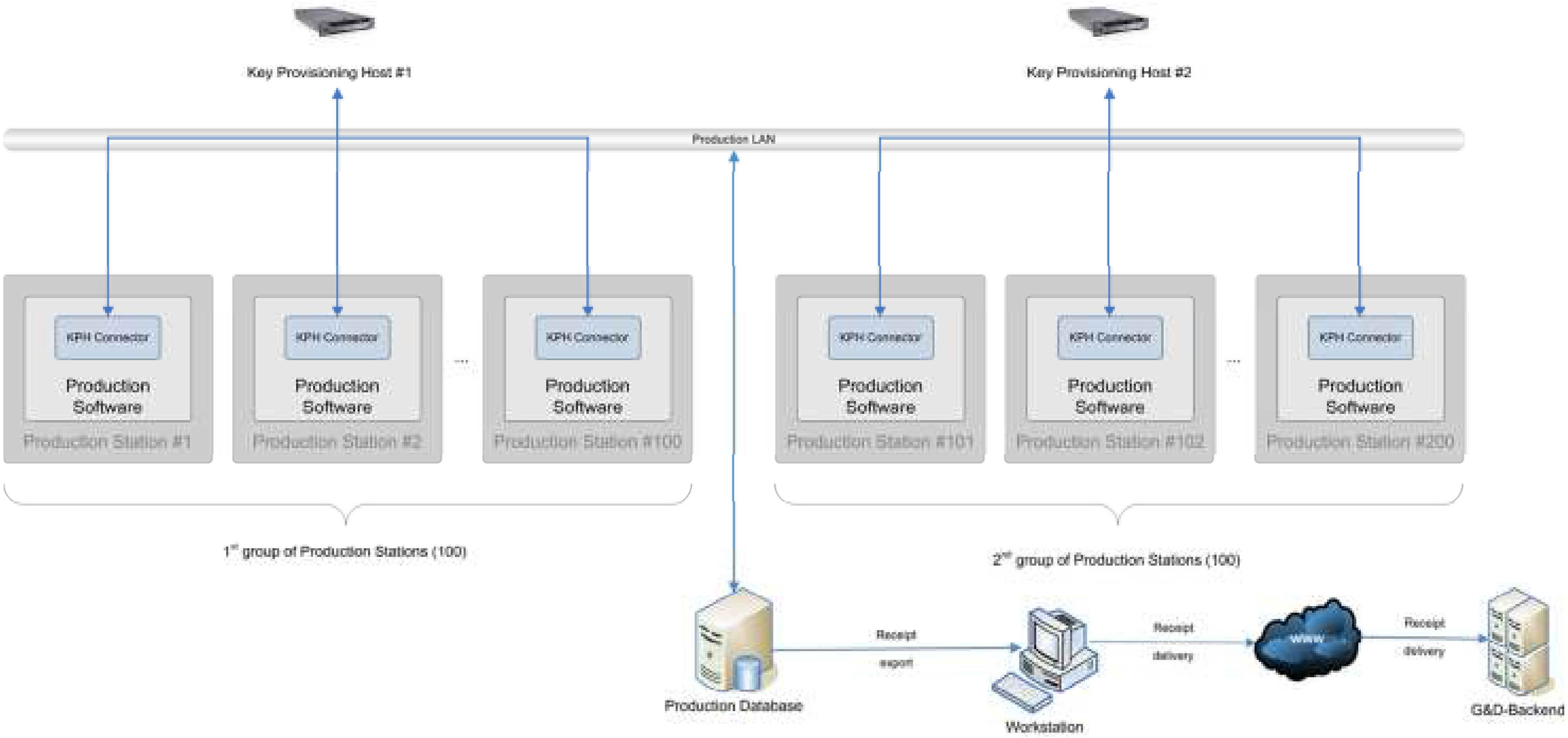

Figure 2: Multiple Key Provisioning Hosts in the Production LAN.

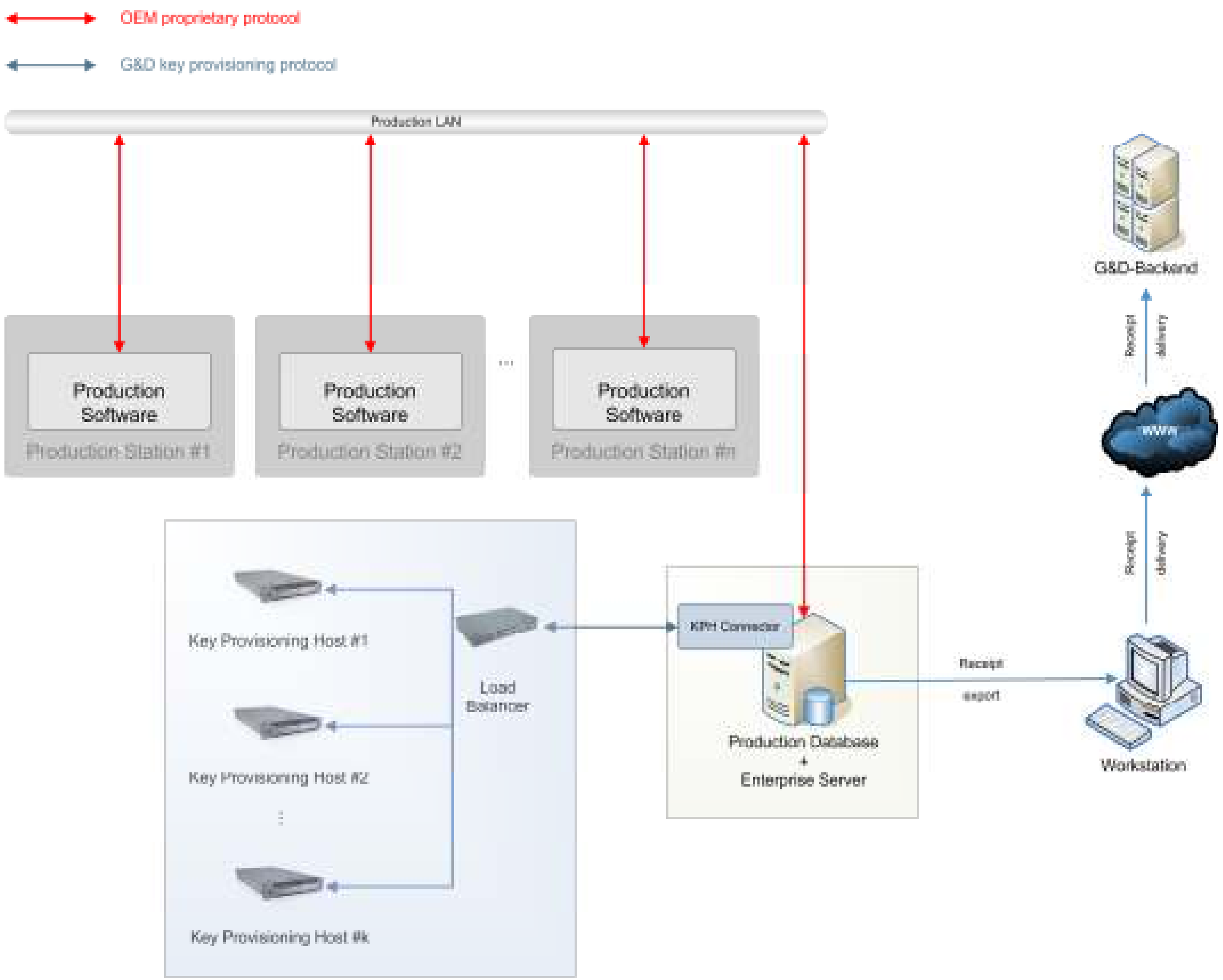

Figure 3: Alternative OEM configuration: Access to the Key Provisioning Hosts (centralized architecture).

Figure 2 illustrates the maximum number of Production Stations per Key Provisioning Host: One KPH is able to handle up to 100 Production Stations[3].

Figure 3 shows an alternative OEM configuration, which is called "*centralized KPH access*": Just a single KPH connector is installed on a centralized server, which may act as the Production Database as well as an "*Enterprise Server*" running the (production) business logic.

The Enterprise Server is connected to a load balancer that in turn connects to several Key Provisioning Hosts. In this scenario, the Key Provisioning Protocol packets are "*tunneled*" from the KPH cluster over the Enterprise Server (running the KPH Connector) to the Production Stations, which transmit them via USB to the attached Mobile Devices.

1.4.2 The Provisioning Library

The Provisioning Library is the communication partner of the KPH Connector during the device binding. It delivers the SUID as well as the authentication token SO.AuthToken to the KPH Connector.

The KPH Connector provides the 256bit AES key K.SoC.Auth, which is generated by the Key Provisioning Host, to the Provisioning Library.

Important note:

Both entities, the **Provisioning Library** (platform: ARM) and the **KPH Connector** (platform: X86/X86-64 on Windows/Linux) are represented by the Provisioning API

[3] Assumption: A maximum number of 16 mobile devices is connected to one Production Station concurrently.

described in this document. The common term "***Provisioning API***" is used to reference both entities.

1.4.3 KPHs in a load balanced environment

For various reasons, the OEM shall think about establishing load balancers with multiple attached KPHs:

- **Business continuity**: If a single KPH fails, then production continues because remaining KPHs behind load balancer handle the requests for the faulting KPH.
- **Maintainability**: If KPHs require a firmware update, then the KPHs can be upgraded one after the other without business interruption.
- **Centralized KPH access and easy configuration**: All KPHs behind the load balancer build a cluster of KPHs, which is accessible by the KPH connector on all Production Stations via a single IP address.

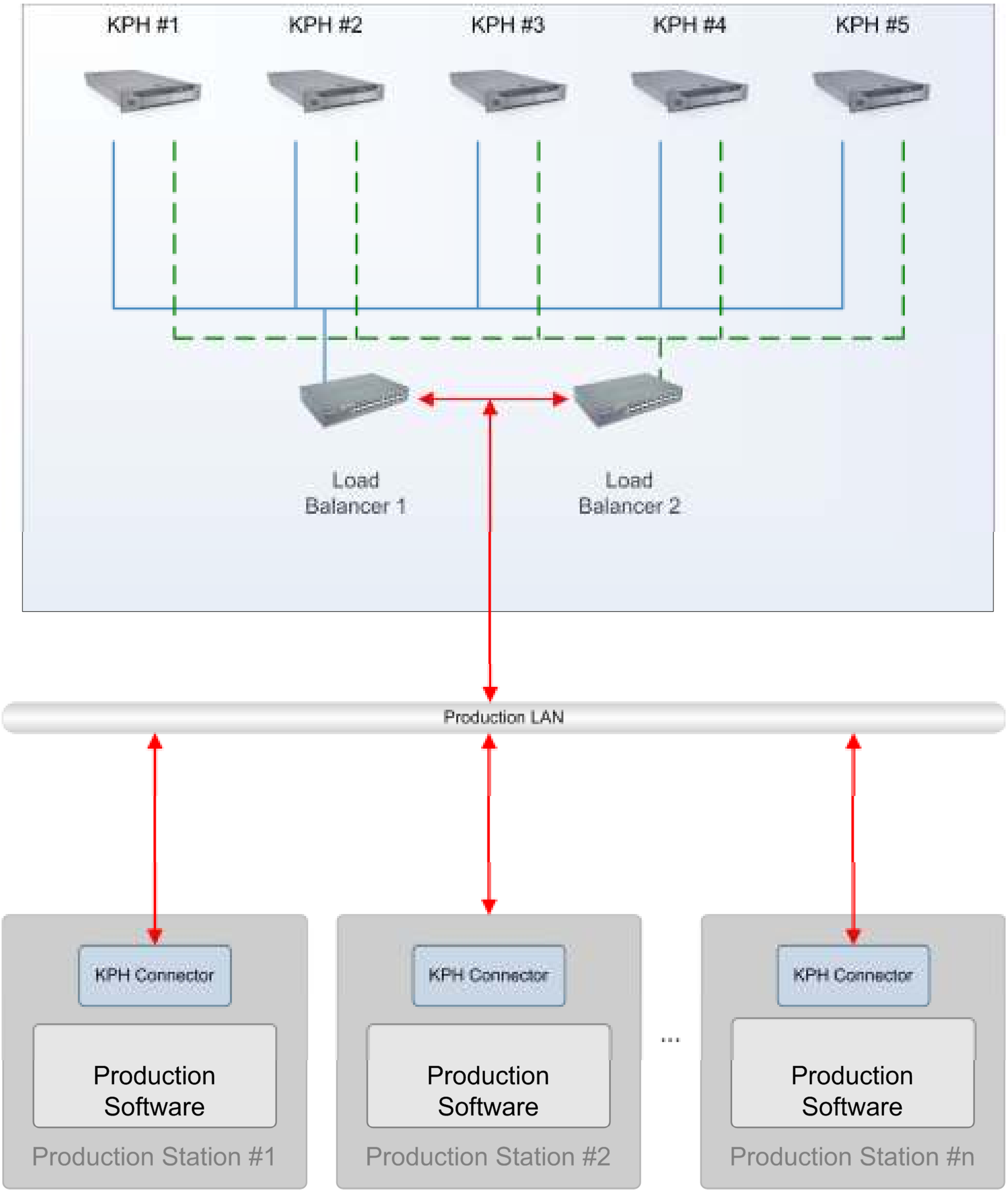

Figure 4: Multiple KPHs behind one or more load balancers.

Figure 4 illustrates an example: Here, two load balancers (for redundancy reasons) connect to all (five) KPHs in the enterprise. In this scenario, up to four KPHs and up to one load balancer may fail without interrupting the key provisioning system as a whole.

The KPH firmware is able to handle this scenario as well. Please refer to the KPH manual **Error! Reference source not found.**.

1.5 KPH INTEGRATION PREREQUISITES

This section enumerates the prerequisites for the integration of the provisioning API in the production environment of an OEM.

1.5.1 Hardware/platform prerequisites

1.5.1.1 Device

An ARM-based device is required.

1.5.1.2 Communication channel (Device ↔ Production Station)

The OEM has to setup and operate a communication channel between the Device and the Production Station for the exchange of communication data packets (the cryptographic provisioning protocol). In the vast majority of the existing production environments, this communication channel already exists and is employed by the standard production process.

The type of this communication channel is beyond the scope of this document. Furthermore – for the Provisioning API – this channel is fully opaque.

Appendix A.1 (on page 53) summarizes some information applicable to USB connections.

1.5.1.3 Production Station

The Production Station is a PC-style host. The platform can be either x86 or x86-64.

~~*One USB 2.0 port shall be available.*~~

1.5.1.4 Receipt Storage

Please refer to subsection 1.5.2.3 on page 21, too.

If **no** database is available for storing receipts, then the Receipt Storage shall be any kind of Network Attached Storage (NAS) or a file server providing a network file system (e.g. SMB, CIFS, NFS).

Section 3.1 (on page 39) is dedicated to the file-based storage of receipts.

1.5.2 Software prerequisites

1.5.2.1 Device

It shall run Android 2.2 (*Froyo*), 2.3.x (*Gingerbread*), 3.x (*Honeycomb*) or 4 (*Ice Cream Sandwich*).

If Android is not available, then additional technical negotiations between the OEM and the Vendor (i.e. G&D) are required.

Furthermore, a dedicated flash storage location for the storage of the SO.AuthToken (152 Bytes) has to be provided.

Depending on the OEM version of the Provisioning Library, the storage of the SO.AuthToken can be performed by either the library (internally) or the OEM. G&D provides this information on a per-OEM basis.

1.5.2.2 Production Station

The Production Station is running either MS Windows (XP, Vista, 7) or Linux (Debian-based or RHEL-based).

1.5.2.3 Receipt Storage

Please refer to subsection 1.5.1.4 on page 20, too.

If a database is available for storing receipts, then either an existing table has to be extended or a new table has to be added to the existing database scheme.

Section 3.2 (on page 39) is dedicated to the database-based receipt storage.

1.6 MOBICORE® CONTENT MANAGEMENT EMULATOR

To be fully independent of the MobiCore® device integration itself, secunet AG implemented the "*MobiCore® Content Management Emulator*" that enables the OEM to convert arbitrary devices in "*MobiCore®-enabled devices*".

Important notes:

1. Please do not be confused of the term "*MobiCore®-enabled devices*". The emulator just emulates the MobiCore® System Trustlet® "CM" and generates an SUID and the Fuse key K.Device.Fuse on request using a pseudo random number generator.
2. The emulator is **not** part of the official deliverables, i.e. the OEM does not get support for the emulator.

Figure 5 illustrates the normal MobiCore® Device fully equipped with the MobiCore® TEE and all required shared objects:

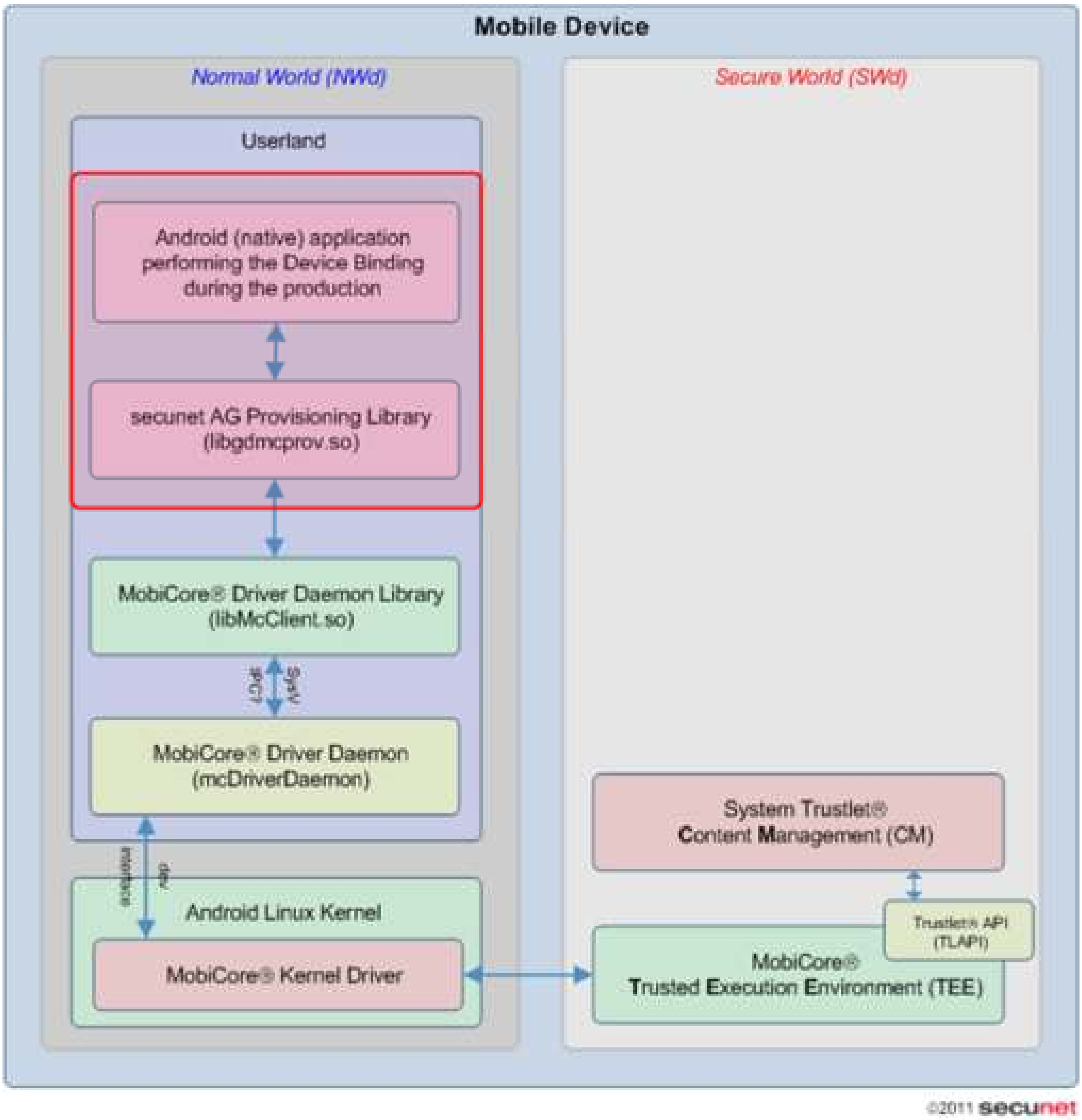

Figure 5: The full MobiCore® architecture as required by the Key Provisioning.

Beginning with the upper, left block in the figure, which is the Key Provisioning Android application, you can see the full stack of software components required in the userland of the normal world:

- **libgdmcprov.so**: Key Provisioning library – the API this document specifies;

- **libMcClient.so**: MobiCore® driver library – accesses the MobiCore® driver daemon;
- **mcDriverDaemon**: MobiCore® driver (UNIX) daemon – communicates with the MobiCore® kernel driver

The mcDriverDaemon is the userland part of the MobCore® kernel driver. It uses special ARM instructions of the ARM TrustZone™ to communicate with the MobiCore® TEE in the SWd. The MobiCore® Content Management emulator is shown in Figure 6:

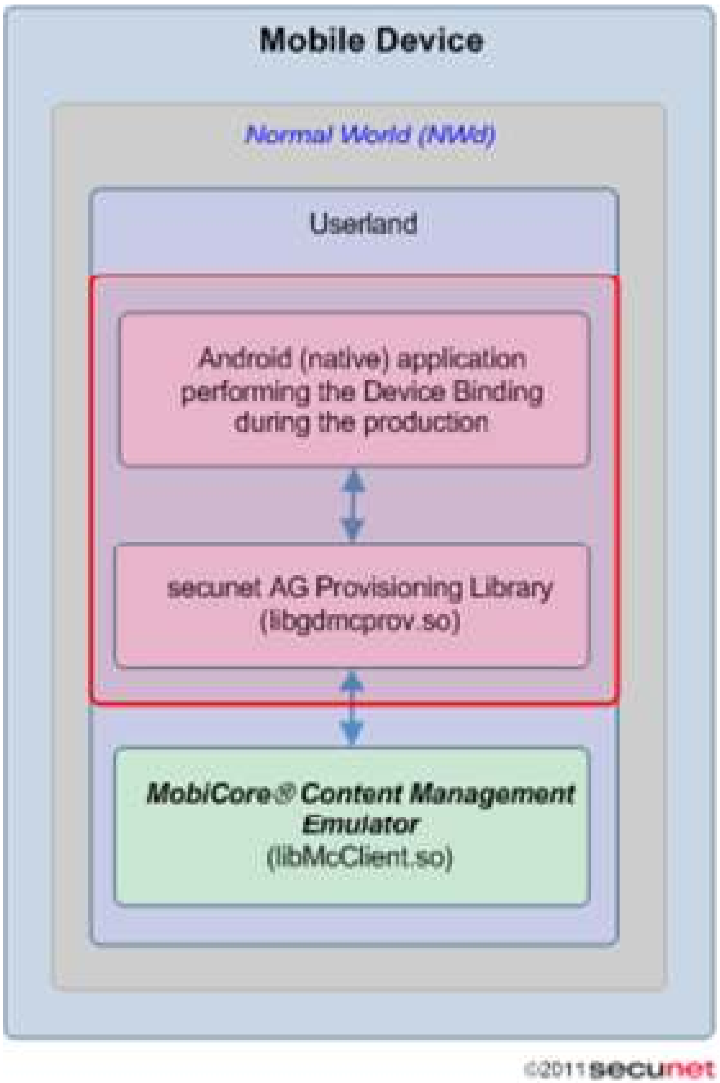

Figure 6: MobiCore® Content Management Emulator.

As you can see, the standard *libMcClient.so* is replaced by the emulator instance of the library. This emulator library is part of the "*MobiCore® Key Provisioning SDK*". Please refer to appendix A.4 (page 68).

The emulator requires the existence of the path:

`/data/app/mcRegistry`

having the access rights[4] **`0777`**. This folder is used to create and access the two files "*suid*" and "*k_device_fuse*" containing the randomly generated SUID and the key K.Device.Fuse.

[4] Please note that the OEM may choose to limit the rights to 0775. In this case, the Android (Dalvik) apps requiring access to the MobiCore® Registry have to be digitally signed by the OEM to allow rights elevation.

To use the emulator, please **do** assign the rights 0777.

2 APPLICATION PROGRAMMING INTERFACE

2.1 OVERVIEW

The Provisioning API is a lightweight API that hides most of the provisioning details from the caller. The API functions are identical for the Production Station and the Device. This yields a homogeneous programming environment for both communication endpoints. Nevertheless, some of the API functions are only available on the Production Station or the Device, respectively.

The Production Station version of the Provisioning API is available as:

- a Windows Dynamic Link Library (DLL), 32bit, Intel x86
- a Windows DLL, 64bit, Intel x86-64
- a Linux Shared Object (so), 32bit, Intel x86 (*upon request*)
- a Linux so, 64bit, Intel x86-64 (*upon request*)

The Device version of the Provisioning API is available as a shared object targeted to ARM (32bit).

Beginning with subsection 2.4 (page 28), some API functions are accompanied by a table summarizing the performed actions (provided for informational purposes). These tables are partitioned according to the Provisioning API versions:

- **Windows/Linux (Prototype)**: All Windows/Linux versions (see above) as implemented by the Provisioning API prototype (no Key Provisioning Host involved);
- **Windows/Linux (Product)**: All Windows/Linux versions (see above) as implemented by the Provisioning API product version (Key Provisioning Host involved);
- **ARM**: The Device version of the Provisioning API; no prototype/product version is discriminated.

The MS Windows versions of the library are equipped with structured exception handlers that catch CPU exceptions improving the runtime stability dramatically.

Moreover, the MS Windows versions perform additional checks for all memory buffers passed as parameters to the API functions for proper read or read/write access, respectively.

All API functions are **MT-safe**.

Appendix A.2 on page 55 contains the C header file of the Provisioning API.

2.2 CONFIGURATION OF THE PROVISIONING API

The Provisioning API requires a certain amount of configuration data (e.g. the IP address of the Key Provisioning Host). To simplify the integration process for the OEMs, the configuration of the Provisioning API library is performed by an external configuration tool, i.e. the actual configuration is patched in the library binary.

[*This tool is not required for the prototype and will be available later on.*]

Note:

Several reviews of this API specification resulted in the inclusion of the new API function `GDMCProvSetConfigurationString`. In a specific OEM environment, it might not be suitable to patch e.g. an IP address in the Provisioning Library binary because the distribution of the resulting (patched) binaries to the Production Stations might not be feasible.

To circumvent this problem, the new API function can be used by the Production Software to configure the Provisioning Library via one configuration string that is comprised of the concatenation of all required configuration information. This string can be stored in a Production Database, too.

2.3 THE CONTROLLING PROCESS

The Provisioning API requires two communication partners: The first one is the Production Station software running on the Production Station. The second one is an executable unit running on the Device, e.g. a small native Linux (Android) application.

Both communication partners must be connected via a transport mechanism, which is fully opaque to the Provisioning API. This is normally a USB connection or a serial connection.

For the rest of this chapter the terms "*process*", "*process context*", and "*controlling process*" denote the controlling application that links to the Provisioning API library on the Device.

2.4 GLOBAL LIBRARY INITIALIZATION AND CLEANUP

2.4.1 Library initialization

The Provisioning API has to be initialized once in process context:

```
gderror GDPROVAPI GDMCProvInitializeLibrary ( void );
```

This function has to be called **once** by the Production Station software or the provisioning process of the Device, respectively.

API version:	Summary of performed actions:
Windows/Linux (Prototype)	• Global initialization (e.g. creation of synchronization objects) • Reading and initialization of RSA keys (for SD.Receipt) • Seeding of PRNG
Windows/Linux (Product)	• Global initialization (e.g. creation of synchronization objects) • KPH initialization • Establishment of TLS-secured channel between Production Station and Key Provisioning Host
ARM	• Global initialization

2.4.2 Library cleanup

To free all resources associated with the Provisioning API, the controlling process has to perform this final call:

```
gderror GDPROVAPI GDMCProvShutdownLibrary ( void );
```

API version:	Summary of performed actions:
Windows/Linux (Prototype)	• Cleanup and freeing of all resources • Termination of all running threads associated with the library
Windows/Linux (Product)	• Cleanup and freeing of all resources • Termination of all running threads associated with the library • KPH cleanup
ARM	• Global cleanup

2.4.3 Callback functions (Device only)

The OEM may provide two callback functions used to write (*device binding*) or read (*device binding validation*) the authentication token SO.AuthToken, respectively:

```
typedef gderror (*authtok_writecb)( const _u8 *authtok,
                                    _u32      authtok_size );

typedef gderror (*authtok_readcb)( _u8  *authtok,
```

```
                                    _u32 *authtok_size );

gderror GDPROVAPI GDMCProvSetAuthTokenCallbacks (
                  authtok_writecb writefunc,
                  authtok_readcb  readfunc );
```

Right after the global initialization of the library, the OEM shall call `GDMCProvSetAuthTokenCallbacks` to provide two function pointers to the write and read functions.

Because the (secure) storage of the authentication token SO.AuthToken is highly OEM-specific, this functionality has to be added by these two hook functions.

As already mentioned in subsection 1.5.2.1 on page 20, the storage of the SO.AuthToken is either performed by the Provisioning Library (internally) or by the OEM (hook functions, externally), respectively. This depends on the delivered Provisioning Library version. The function pointer parameters of GDMCProvSetAuthTokenCallbacks can be `NULL` to indicate that the respective hook function is not implemented.

Important note:

It is very important that the OEM and the vendor (G&D) agree on which party is responsible for storing SO.AuthToken. In the unlikely event that neither the Provisioning Library nor the Production Software (hook functions) store SO.AuthToken, **all provisioned mobile devices will become useless with respect to MobiCore®**.

2.5 PERFORMING THE DEVICE BINDING

The Provisioning API library is fully MT-safe. A Production Station normally concurrently connects to several devices to be provisioned. The Production Station software has to create a dedicated "*device binding instance*" for each device connected to the Production Station. This can be done either from one execution thread or a dedicated thread can be created for each device calling the function in subsection 2.5.1 once in each thread, respectively. It is up to the OEM (and up to the design of the Production Station software) to decide which architecture (single-threaded vs. multi-threaded) fits better.

2.5.1 Creating one device binding instance

For each device to be provisioned, one dedicated device binding instance has to be created:

```
gderror GDPROVAPI GDMCProvBeginProvisioning ( gdhandle *provhandle );
```

If the function returns signaling success, then `provhandle` is filled with a handle to a newly created device binding instance.

2.5.2 Executing the device binding protocol

To simplify the integration of the key provisioning ("*device binding*") in the Production Station software, one single function is provided by the Provisioning API library, which has to be called several times in a row. The architecture of the Provisioning API defines an internal DFA ("***D**eterministic **F**inite **A**utomata*") that keeps track of the current device binding stage.

```
gderror GDPROVAPI GDMCProvExecuteProvisioningStep (
                  gdhandle     provhandle,
                  const _u8   *msgin,
                  _u32         msgin_size,
                  _u8         *msgout,
                  _u32        *msgout_size );
```

API version:	Summary of performed actions:
Windows/Linux (Prototype)	• Full implementation of the cryptographic protocol (device binding) • Request of SUID (1st message) • Generation of random AES-256bit key (PRNG) • Request of authentication token (2nd message) • Creation of the receipt SD.receipt • Request of the SO.AuthToken validation (3rd message)
Windows/Linux (Product)	• Full implementation of the cryptographic protocol (device binding) • Request of SUID (1st message) • Generation of random AES-256bit key by the TRNG of the Key Provisioning Host • Request of authentication token (2nd message) • Creation of the receipt SD.receipt delegated to the Key Provisioning Host • Request of the SO.AuthToken validation (3rd message)
ARM	• Full implementation of the cryptographic protocol (device binding) • Delivery of SUID (1st message) • Creation, delivery, and storage of the authentication token SO.AuthToken (2nd message) • Read-back and validation of SO.AuthToken upon request (3rd message)

This function has to be called in a loop. The previous received message from the communication partner (Device for the Production Station or Production Station for the Device, respectively) is `msgin` or `NULL` if the first message has to be generated on the Production Station.

The next message to be exchanged is stored in the buffer denoted by `msgout`. The return value of the function signals the completion of the entire device binding process.

The following code snippet illustrates the device binding sequence for the **Production Station**: This code was taken from the implementation of the Production Station mock-up (demonstration software) that is listed in appendix A.4.5.1 (on page 73).

```
  // 2.) Perform provisioning loop

  msgin_size  = 0; // signal no previous message available
  msgout_size = sizeof(msgout); // initialize with available bytes

  fprintf(stdout,"Entering provisioning loop.\n");

  while (GDERROR_OK==(err=GDMCProvExecuteProvisioningStep(
         provhandle,msgin,msgin_size,msgout,&msgout_size)))
  {
    // send message to device (if available)

    if (0!=msgout_size)
    {
#ifdef _DEBUG
      fprintf(stdout,"SEND TO DEVICE: %u byte(s):\n",msgout_size);
      gdmc_hexdump(msgout,msgout_size);
#endif
      if (!comm_send(msgout,msgout_size))
      {
        fprintf(stderr,"ERROR: send to device failed.\n");
        err = GDERROR_UNKNOWN; // use unknown error code to signal transmissione error
        break;
      }
    }

    // receive next message from device

    msgin_size = sizeof(msgin);
    if (!comm_recv(msgin,&msgin_size))
    {
      fprintf(stderr,"ERROR: recv from device failed.\n");
      err = GDERROR_UNKNOWN; // use unknown error code to signal transmissione error
      break;
    }
#ifdef _DEBUG
    fprintf(stdout,"RECV FROM DEVICE: %u byte(s):\n",msgin_size);
    gdmc_hexdump(msgin,msgin_size);
#endif

    // Check if we have to abort the provisioning loop

    if (GDERROR_OK!=err)
      break;

    msgout_size = sizeof(msgout); // initialize with available bytes (for next iteration)
  }
```

The initialization of the library and the creation of one device binding instance are not shown here. The core provisioning loop consists of an outer loop that is performed several times (e.g. five times). This is done due to the possibility that one of the provisioning tries might fail. The Provisioning Library can handle and recover from such an error condition without the need to free and re-acquire a (new) provisioning handle.

The provisioning loop itself (the "`while`" compound block) calls `GDMCProvExecuteProvisioningStep` passing the recently received message as the "*in-message*" and providing an empty buffer for the next "*out-message*" to be sent to the communication partner (here: the Device). After that, the *out-message* is sent to the Device (by calling `comm_send`). Then, `comm_recv` is called to receive the response (*in-message*) for the recently sent command message (*out-message*).

The provisioning loop is left if and only if `GDMCProvExecuteProvisioningStep` returns an error code other than GDERROR_OK. Three possible cases have to be handled now:

1. Recent result code was `GDERROR_PROVISIONING_DONE`: This means that the key provisioning and the validation of the provisioning process were successful. The provisioning loop is aborted.
2. Recent result code was `GDERROR_VALIDATION_FAILURE`: This means the the key provisioning was successful but the validation of the provisioning process failed. The provisioning loop is aborted.
3. Any other result code: An error occurred (e.g. CRC32 error). The outer loop is not left and the next provisioning try is performed.

In the second case, the OEM may decide to either perform another provisioning try or to tag the actual device as "broken".

The device binding code sequence for the Device is slightly different: The main loop shall start with a call of the *receive* function. Once the first message has been received from the Production Station, `GDMCProvExecuteProvisioningStep` is called. A new `msgout` is generated, which has to be transferred by the *send* function back to the Production Station.

The source code in appendix A.4.5.2 (on page 78) illustrates the full implementation.

2.5.2.1 Getting the result of the device binding

The code snippet in the previous subsection (2.5.2) does not show the storage/processing of the final result.

For the Production Station, the final `msgout` (provided when `GDMCProvExecuteProvisioningStep` returns `GDERROR_PROVISIONING_DONE`) contains the receipt SD.Receipt. For the Device, the final `msgout` is empty.

The write callback function (please refer to subsection 2.4.3 on page 28) is called on the Device side to actually store the authentication token.

The Production Station version of the Provisioning API exports two additionnal functions to support the processing of the SD.Receipt:

1. `GDMCProvFormatReceipt` (generation of a BASE64 encoding of the binary SD.Receipt); please refer to subsection 2.5.4.2 on page 36
2. `GDMCProvGetSUID` (retrieval of the SUID assigned to the SoC of the current Device); please refer to subsection 2.5.4.5 on page 36

Ideally, the generated SD.Receipt is written back into the Production Database. Because handling BLOBs (Binary Large Objects) can add more complexity to a database, binary data is often converted to the BASE64 representation and added to the database as a normal string (varchar). `GDMCProvFormatReceipt` delivers the BASE64 encoding for a binary data bucket. `GDMCProvGetSUID` returns the SUID (as a binary array of 16 octets) that will be added most likely to the database, too.

2.5.2.2 More details about error handling

The device binding process involves two parties:

1. The Production Station, which is the **initiator**.
2. The Device, which is the **responder**.

The initiator acts as the **master**, the responder as the **slave** in this communication scenario.

The API function `GDMCProvExecuteProvisioningStep` is designed in a way that it can handle unexpected input messages resulting from erroneous control flows.

<u>Example:</u>

The Production Station generates the first message and calls `TransferMessageToCommPartner`. The Device receives the first message, generates a response and sends this response back to the Production Station.

The response is not received by the Production Station for any reason. In this case, `TransferMessageToCommPartner` returns an error (e.g. due to a timeout).

The Production Station software can either destroy and re-create the device binding handle in this case (`GDMCProvEndProvisioning` followed by `GDMCProvBeginProvisioning`) or just re-enter the provisioning loop (the internal state of the library ensures proper error recovery) thus performing a retry of the device binding process.

Again, the first message is (re-)created by the Production Station and sent to the Device by calling `TransferMessageToCommPartner`. The Device receives this first message (again) although it expects the second message of the device binding process because it successfully processed the first one before and has no chance to recognize errors occurred at the Production Station.

This situation does not result in an error on the Device side, because the API function `GDMCProvExecuteProvisioningStep` treats this repeated (first) message as a communication error and (re-)generates the response for the first message again.

As a rule of thumb, the responder (the Device) should always act in a fault tolerant way. Any errors occurring in the Device should result in an optional re-creation of the device binding handle followed by the (intial) call to the *receive* function (in all cases).

The initiator acts similarly, i.e. a retry is initiated by an internal restart of the entire device binding process. Together with the built-in fault tolerance of the API function `GDMCProvExecuteProvisioningStep`, there is a high probability that both communication partners re-synchronize after an error occurred on either side.

2.5.3 Extra validation of the device binding (optional)

<u>Important note:</u>

The following API function has been removed from the API because the extra validation step is now a mandatory part of the provisioning loop.

~~The Provisioning API function:~~

```
gderror GDPROVAPI GDMCProvExecuteValidationStep (
                  gdhandle    provhandle,
                  const _u8   *msgin,
                  _u32        msgin_size,
                  _u8         *msgout,
                  _u32        *msgout_size );
```

~~can be used by the Production Station and the Device to add a secondary loop after the provisioning loop terminated successfully. Again, the communication is initiated by the Production Station, which acts as the initiator; the Device continues to act as the responder.~~

~~The first (and only) message generated by `GDMCProvExecuteValidationStep` (Production Station) sends SO.AuthToken to the Device to trigger a read-back of the authentication token SO.AuthToken and to compare these two binary data buckets. The result is sent back to the initiator.~~

~~Furthermore, the Production Station's version of `GDMCProvExecuteValidationStep` validates the receipt SD.Receipt locally[5]. The extracted authentication token SO.AuthToken is the token sent to the Device for comparison.~~

~~After this simple message exchange, both parties "know" if the device binding and the storage of the authentication token in the device were successful.~~

~~The API function `GDMCProvExecuteValidationStep` is used in exactly the same way as `GDMCProvExecuteProvisioningStep`, i.e. in a loop. The final error code (aborting the loop) can be one of:~~

1. ~~`GDERROR_VALIDATION_SUCCESSFUL`: signaling a successful validation process;~~
2. ~~`GDERROR_VALIDATION_FAILURE`: signaling a validation failure;~~
3. ~~Any other code except for `GDERROR_OK`: other error occurred.~~

~~`GDERROR_OK` signals to continue the validation loop.~~

2.5.4 The remaining API functions

2.5.4.1 Releasing a device binding instance (handle)

The API function:

```
gderror GDPROVAPI GDMCProvEndProvisioning ( gdhandle provhandle );
```

frees all internal resources associated with one device binding instance.

[5] Please recall that SO.AuthToken is part of the receipt SD.Receipt.

2.5.4.2 Formatting the receipt

The API function:

```
gderror GDPROVAPI GDMCProvFormatReceipt (
                  const _u8   *receipt,
                  _u32         receipt_size,
                  _u8         *fmt_receipt,
                  _u32        *fmt_receipt_size,

                  _u8         *suid );
```

is only available in the Production Station version of the Provisioning API library.

The Production Station software uses this function to create a BASE64-encoded version of the receipt SD.Receipt and to return the SUID of the Device's SoC as a sequence of 16 bytes.

Please refer to chapter 3 beginning on page 39 for more information about the receipt handling.

2.5.4.3 Getting the version of the Provisioning API

The process can query the version of the Provisioning API by calling:

```
_u32 GDPROVAPI GDMCProvGetVersion ( void );
```

The version number is split into four parts (one byte each):

```
major | minor | patch level | OEM ID
```

The version number 1.2.3.4 represents the major version 1, minor version 2, patch level 3 (revision, bug-fixes), and the OEM ID, which is statically assigned by G&D (here: OEM 4).

2.5.4.4 Retrieving the error message for an error code (Production Station)

The API function:

```
gderror GDPROVAPI GDMCProvFormatErrorMessage ( gdhandle provhandle,
                                              gderror  errorcode,
                                               char    *msgbuf,
                                              _u32     *size );
```

dumps a detailed error message to the supplied buffer for a specified error code. If more detailed information about the (recent) error is available, then this information is added as well.

Please note that all messages are UTF-8 encoded to ease the localization of the error messages. Currently, all messages are dumped in English only.

2.5.4.5 Retrieving the SUID of an SoC (Production Station)

The API function:

```
gderror GDPROVAPI GDMCProvGetSUID (
                        gdhandle    provhandle,
                        _u8        *suid );
```

delivers the SUID of the SoC to the caller. The function fails if the SUID is not (yet) known. The SUID is available after the first message exchange. Because the caller cannot determine this point in time for sure, it is recommended to call this function only after performing the entire provisioning loop.

2.5.4.6 Configuring the Provisioning Library (Production Station)

The API function:

```
gderror GDPROVAPI GDMCProvSetConfigurationString (
                         const char *config_string );
```

can be used on the Production Station to pass a zero-terminated configuration string to the Provisioning Library. The format of this string (if any) has to be negotiated between G&D and the OEM on a per-OEM basis.

Table 1 enumerates all configuration values that are currently available:

Configuration item:	Allowed values:	Description:
GDMCPROVLIB_SOCKET_ACQUIRE_TIMEOUT	1..180	timeout value (in seconds) the KPH connector is trying to acquire a secured connection to the KPH
GDMCPROVLIB_KPH_SERVICE	*IP*:*PORT*	IP address and TCP port of the KPH
GDMCPROVLIB_SOCKET_POOL_SIZE	1..1000	Number of concurrent connection between the KPH connector and the KPH (pooled)

Table 1: KPH configuration values.

Examples (KPH configuration strings):

Example #1: Single connection

```
GDMCPROVLIB_SOCKET_ACQUIRE_TIMEOUT=5;
GDMCPROVLIB_KPH_SERVICE=192.168.76.2:9910; GDMCPROVLIB_SOCKET_POOL_SIZE=1
```

Just one connection can be established from the Production Station to the Key Provisioning Host at a time.

Example #2: Multiple concurrent connections

```
GDMCPROVLIB_SOCKET_ACQUIRE_TIMEOUT=180;
GDMCPROVLIB_KPH_SERVICE=192.168.76.2:9910; GDMCPROVLIB_SOCKET_POOL_SIZE=16
```

A Production Station may open 16 concurrent connections to the Key Provisioning Host in parallel.

Example #3: Multiple concurrent connections with multiple KPHs behind a load balancer

```
GDMCPROVLIB_SOCKET_ACQUIRE_TIMEOUT=180;
GDMCPROVLIB_KPH_SERVICE=192.168.76.2:9910; GDMCPROVLIB_SOCKET_POOL_SIZE=64
```

A Production Station may open 64 concurrent connections to the Key Provisioning Host in parallel. In this example, four KPHs are available behind a load balancer so that 16 connections per KPH are established for load balancing and higher availability.

2.6 THE MESSAGE FORMAT

The cryptographic protocol that is performed between the Production Station and the Device is outlined in appendix A.3.1 (on page 63).

Because the OEM-specific implementation of the USB communication between the Production Station and the device(s) may need to compute the full length of a message based on the message header, these internal C structures are declared in the public header file `gdmcprovlib.h`, too.

The footer section of the C header file declares the message header and the message trailer:

```
typedef struct _gdmc_msgheader          gdmc_msgheader;
typedef struct _gdmc_msgtrailer         gdmc_msgtrailer;

/// the G&D MobiCore message header
struct _gdmc_msgheader
{
  _u32        msg_type;   ///< message type
  _u32        body_size;  ///< size of body (may be 0)
} PACK_ATTR;

/// the G&D MobiCore message trailer
struct _gdmc_msgtrailer
{
  _u32        magic;      /// message type (one's complement)
  _u32        crc32;      /// CRC32 checksum

} PACK_ATTR;
```

An OEM-specific receive function (USB) can read the first eight octets (containing two _u32 values). The second _u32 value (`body_size`) can be used to compute the remaining size of the message:

```
remaining = header->body_size + sizeof(gdmc_msgtrailer)
```

The variable "`remaining`" denotes the size of the message in bytes excluding the message header (eight bytes). The receive function can then try to receive the remaining data of the current message in a second step. The source code "`tcpipnetworking.c`" contains sample code (for a TCP/IP-based communication) that is implemented in exactly this way.

Please note that all message values that a larger than one octet (byte) are transmitted in *Little Endian* byte order. This is due to the fact that all supported platforms (x86, x86-64, and ARM) are Little Endian machines. There is no need to convert all values first to network order (which is Big Endian) and then back to Little Endian.

3 RECEIPT STORAGE AND TRANSFER

The Provisioning API generates one receipt for each provisioned device, denoted by SD.Receipt. This is a binary data bucket that has to be transferred to the G&D backend, e.g. by E-mail or via RSync/SSH (please refer to subsection 3.5.1 on page 46).

Depending on whether the receipts are stored in a network storage or in a central database, a small tool (e.g. a shell script) is required to export (bundle) these receipts in a so called "*receipt log file*", which is text-based (please refer to section 3.3 on page 40).

This receipt log file has to be sent to G&D. G&D imports the contained data in the G&D backend so that the other MobiCore® use cases can be performed between the Device and the G&D Backend System.

3.1 FILE-BASED RECEIPT STORAGE

The OEM shall establish a network storage that is accessible by all Production Stations in the production network. A dedicated receipt SD.Receipt can be formatted by using the API function `GDMCProvFormatReceipt` (please refer to subsection 2.5.4.2 on page 36).

The result of this function is the data bucket SD.Receipt (BASE64-encoded); the SUID (binary, 16 bytes) can be queried by calling `GDMCProvGetSUID` (please refer to subsection 2.5.4.5 on page 36). The OEM shall add additional information to these two items to create a string according to section 3.3 on page 40. This string can directly be appended to the receipt log file or temporarily stored in separate files, which have to be consolidated in one big file (the receipt log file).

3.2 DATABASE-BASED RECEIPT STORAGE

If a production database is in-place, then it can be modified to include the additional information delivered by the device binding process.

At the convenience of the OEM, either an existing database table may be extended or a new database table can be created.

The OEM should call `GDMCProvFormatReceipt` to get the BASE64-encoded SD.Receipt and `GDMCProvGetSUID` to retrieve the binary SUID. It is then up to the OEM to store the receipt either as a BLOB (binary) or as a (VAR-) CHAR (BASE64) along with the SUID in the database. If necessary, then the SUID can also be converted to a textual representation, e.g. as 32 hexadecimal digits representing the 16 binary octets.

As already mentioned in section 3.1 (file-based receipt storage), a small tool (aka shell script) is required to export the data from the database to the receipt log file, which is always one flat (text) file containing the receipts line-by-line. Section 3.4 on page 42 presents a working example.

3.3 THE RECEIPT LOG AND RECEIPT ACKNOWLEDGE FILES

In the following subsections, text-based files are specified that contain multiple datasets line by line. A line is always terminated by the newline character 0x0A ('\n').

3.3.1 Format of the receipt log file

The receipt log file is a text-based file containing receipts (with accompanying data) line by line.

The format of a line in the receipt log file is as follows (all items of a line separated by semicolons or another separator, e.g. '|'):

1. SUID (hexadecimal representation, 32 digits);
2. Refurbishment flag: "0" if new device, "1" if refurbished device
3. SD.Receipt (BASE64-encoded);
4. IMEI of device (integer digits);
5. OEM (identification of OEM in G&D backend, must be a static and a unique identifier);
6. Device model identifier;
7. Operator (if applicable or empty string); applies if and only if this device is branded and dedicated to a specific operator.
8. SiP (Silicon Provider) and SoC model
9. [conditional] more OEM-specific items

3.3.2 Format of the receipt acknowledge file

The receipt acknowledge file is a text-based file containing the SUIDs of the imported SD.Receipts line by line.

The first line of the file contains the global status of the operation. The format of the first text line is:

<status code>;<status message>

The status code is "0" if the operation succeeded for all items or an error code not equal to "0" if the operation failed (partially). The status message is just the string "OK" (plus some information about the number of imported receipts) for the status code "0" or contains a descriptive message of an error that might have been occurred during the import of the data into the G&D Backend System.

The first line is followed by one line for each SD.Receipt that was sent to the G&D Backend System:

<SUID>;<status code>;<status message>

The format of the status code and the status message are identical to the global status (see above) – this code and message reflects the status of the import operation for the current item.

The OEM can use this file to update the Production Database, e.g. to remove unnecessary SD.Receipt entries from the database.

3.3.3 Important remark on error conditions

If the receipt log file matches the specified format, then G&D guarantees that the G&D Backend System import will **never** fail.

Nevertheless, the OEM should be prepared to receive a receipt acknowledge file indicating an error condition. Examples for possible errors include (but are not limited to):

- Format error(s) in receipt log file
- RSA signature validation failed (wrong RSA signature key used to generate SD.Receipt)
- RSA private decrypt failed (wrong RSA encryption key used to generate SD.Receipt)
- Refurbishment flag "0" (new device) but this SUID already known by the G&D Backend System (i.e. database entry exists)

3.3.4 Full list of error codes (acknowledge file)

Error code:	**Meaning:**
0	OK
101	Trailing characters found in the receipt log file (ignored)
102	Receipt log file syntax error (OEM: please cross-check and correct)
103	Number of input line field in the receipt log file mismatches configuration in G&D Backend System (OEM: please contact G&D support)
104	At least one mandatory field was left empty (OEM: please cross-check and correct)
105	At least one input field is bigger than allowed (exceeds limit; OEM: please cross-check and correct)
106	Unable to BASE64-decode receipt in receipt log file (OEM: please cross-check and correct)
107	Validation of RSA signature failed (OEM: please cross-check and contact G&D support for troubleshooting)
108	RSA decryption failed (OEM: please cross-check and contact G&D support for troubleshooting)

109	SUID mismatch of input line and receipt (OEM: please cross-check, this may indicate an error in the receipt log file generation!)
110	An unknown error occurred (OEM: please contact G&D support)
111	Refurbishment flag must be either 0 or 1 (OEM: please cross-check and correct)
112	Unable to update G&D backend database (refurbished device); OEM: This means that the mobile device was tagged "refurbished" but was never imported in the database before
113	Unable to set custom data for the current data item (OEM: please cross-check and contact G&D support for troubleshooting)
114	This MobiCore® mobile device is already in the G&D database and cannot be imported again; OEM: Please cross-check and correct, this may indicate an error in the receipt log file generation!
115	The KPH (Key Provisioning Host) that performed the device binding is not known by Giesecke & Devrient. A non-authorized KPH box was used to perform the device binding. The device binding is rejected by the G&D backend systems.

Table 2: Error codes reported in the receipt acknowledge file.

3.4 A (HYPOTHETIC) DATABASE EXAMPLE

Let us assume that there is already a database table called "*DEVICES*" in the Production Database, e.g.:

```
CREATE TABLE DEVICES (
           IMEI     CHAR(32)    NOT NULL,
           MODEL    VARCHAR(64) NOT NULL,
           OPERATOR VARCHAR(64),
           PRIMARY KEY(IMEI)
      );
```

In this example, only the IMEI of the mobile device, its model and the optional operator[6] is shown.

The existence of another table, "*SD_RECEIPTS*", is assumed:

```
CREATE TABLE SD_RECEIPTS (
```

[6] The optional operator may be used if a specific device model is branded for a specific operator.

```
            SUID        CHAR(32)    NOT NULL,
            SD_RECEIPT CHAR(1024) NOT NULL,
            IMEI        CHAR(32)    NOT NULL,
            EXPORTED    CHAR(1)     NOT NULL,
            PRIMARY KEY(SUID),
            CONSTRAINT FK_IMEI FOREIGN KEY (IMEI) REFERENCES
         DEVICES(IMEI)
      );
```

The SUID is stored as 32 hexadecimal digits (representing the 16 bytes of the SUID). The receipt is stored BASE64-encoded (768 bytes require 1024 characters in this case). The IMEI is added as a foreign key into the DEVICES table. The “EXPORTED” flag concludes the table: It is ‘0’ for a row that was not yet transmitted to G&D and ‘1’ for an already processed item.

Figure 7 illustrates this database schema snippet:

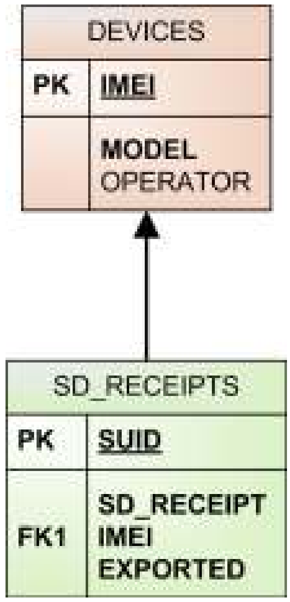

Figure 7: Sample database schema (snippet).

On a regular basis, e.g. daily or when a lot has been produced entirely, a receipt log file has to be generated. To accomplish this task, a temporary database table "*RECEIPT_LOG*" is generated:

```
CREATE TABLE RECEIPT_LOG (
      SUID       CHAR(32)    NOT NULL,
      SD_RECEIPT CHAR(1024)  NOT NULL,
      IMEI       CHAR(32)    NOT NULL,
      MODEL      VARCHAR(64) NOT NULL,
      OPERATOR   VARCHAR(64),
      PRIMARY KEY (SUID)
);
```

The following steps (again, as an example) have to be performed:

1. Insert all new receipts into the temporary table that have not been processed yet:

```
INSERT INTO RECEIPT_LOG
        SELECT SD_RECEIPTS.SUID, SD_RECEIPTS.SD_RECEIPT,
        DEVICES.IMEI, DEVICES.MODEL, DEVICES.OPERATOR
        FROM SD_RECEIPTS, DEVICES WHERE
        SD_RECEIPTS.IMEI=DEVICES.IMEI AND
        SD_RECEIPTS.EXPORTED='0';
```

2. Update the "*EXPORTED*" flag according to the rows of the temporary table:

```
UPDATE SD_RECEIPTS SET EXPORTED='1' WHERE
        SUID IN (SELECT SUID FROM RECEIPT_LOG);
```

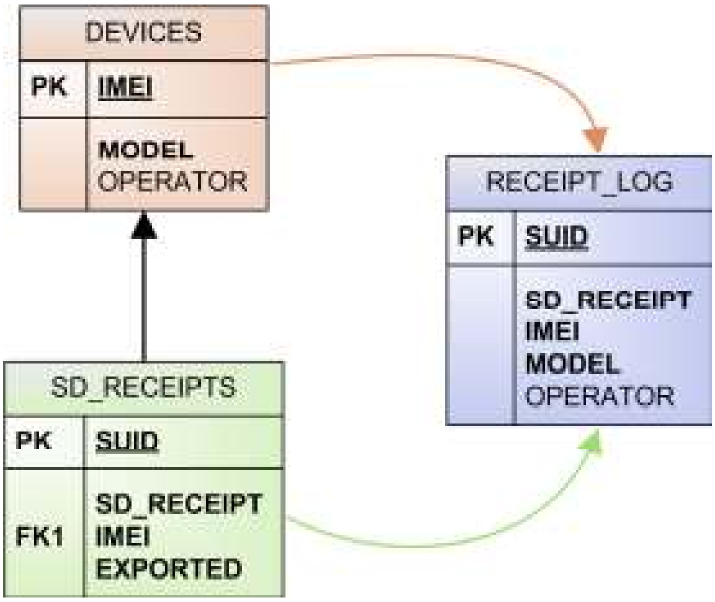

Figure 8: Sample database schema (snippet, continued).

3. Create the receipt log file:

```
SELECT SUID,SD_RECEIPT,IMEI,MODEL,OPERATOR
        FROM RECEIPT_LOG;
```

4. Drop the temporary table:

```
DROP TABLE RECEIPT_LOG;
```

5. Transfer the receipt log file to G&D (please refer to subsection 3.5);

6. Receive the receipt acknowledge file from G&D (please refer to subsection 3.5);

7. Delete rows from the table "*SD_RECEIPTS*" according to the SUIDs of the acknowledge file.

3.5 RECEIPT TRANSFER TO THE VENDOR (G&D)

G&D will negotiate with each OEM how the receipt log file has to be transferred to G&D. This step is always performed fully-automated.

3.5.1 Fully-automated receipt transfer over RSync/SSH

G&D offers a (standard & state-of-the-art) mechanism for the transfer of the receipt log file (OEM → G&D) and for the transfer of the receipt acknowledge file (G&D → OEM).

The mechanism relies on two UNIX tools:

- SSH server (Secure Shell)
- RSync tool (Remote sync)

These two UNIX[7] tools can be combined to setup a very secure bidirectional communication channel between the OEM and G&D.

3.5.1.1 RSA keys

Both communication partners (abbreviated "oem" or "gud", respectively) have to generate an RSA key pair (2048bit). This can be done by entering:

```
ssh-keygen -t rsa -b 2048 -f ./sd_receipt_oem.key ↲
-C '<a comment>' -N ''
```

OR:

```
ssh-keygen -t rsa -b 2048 -f ./sd_receipt_gud.key ↲
-C '<a comment>' -N ''
```

This generates two files for each party: One file contains the RSA private key, the other one the RSA public key (this file is suffixed by "*.pub*").

Each party transfers the RSA public key together with the RSA public **host** key[8] to the opposite party (e.g. via secured E-mail).

3.5.1.2 RSync over SSH

For security reasons, none of the communication partners grants the opposite party the right to perform a login on the host. The incorporation of RSync makes this possible.

Each communication partner creates a new account on the host, e.g. called "*mobicore*". This account creates a home directory for this user as well, e.g. "*/home/mobicore*".

Now, a subdirectory "*.ssh*" has to be created, i.e. "*/home/mobicore/.ssh*". The RSA public **host** key (of the opposite communication partner) has to be added to the file:

---

[7] If a UNIX host is not available, then Cygwin (www.cygwin.com) can be used.

[8] This key is generated automatically by the SSH server during the setup cycle.

/home/mobicore/.ssh/**known_hosts**

The RSA public key file from the opposite communication partner has to be added to the file:

/home/mobicore/.ssh/**authorized_keys**

When an SSH connection is established from the remote host to the local host, a dedicated command to be executed can be added right after the RSA key in the file "*authorized_keys*". This is the key of the secure solution: The communication partner, which is opening the connection, can neither execute an arbitrary command on the remote host nor open a shell. Only the "hard-wired" command in the file "*authorized_keys*" is automatically executed. This command is typically a shell script that employs RSync to pull a file from the remote host through the SSH tunnel to the local host.

Figure 9 illustrates the setup:

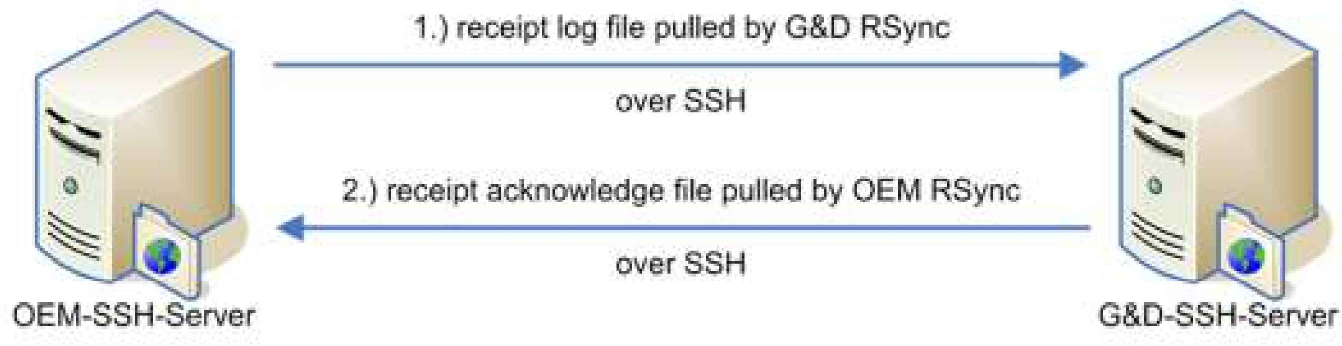

Figure 9: RSync/SSH setup.

3.5.1.3 A working example

The RSA key generation yields two files (example shown for the OEM communication partner):

<u>File "*sd_receipt_oem.key*":</u>


```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEApADnOQYjBGDzdszfv1iP6L9k9m4d0a3xT/DvHT3kH7Uy7Tqu
/K3uTNABvLthN4ZWmzRyTUWxOeI5NHTYpRW1z4c8rn1VRFGOd+tArr+HW4+U5UhS
C6P8Mvb9DolzJ1uPp6sqhe5LpQ/QnE8y5atCaA+Vz1rj63r/ZAoFGrFRDkUgHcO7
nbegJFEheb/st6S2Q1/9NAh6YZcXMZNg7IhOTZHcYhBPyk3FVvVGtLzNiTLyBL5Y
XvPH61tE5YHalAJSB0J5yiaQtAOYmhtpjpAWLqgS4GrqLCGd2k84vcDkPfmVW0U8
8im1k0rlInqyCD1B0RrUTJ4Mzo3ZUsGEm2D9AQIBIwKCAQB51MkFygtiVqY6/peV
dP0wji2+YG3sOBJYpFnbNUpgsn2aSNnDBNzomoTymdMwj64M5SG1vr4q/9K5QNtz
UfTGDLC8Ipcczt7cvWM/+/4mwm6bshEP9ikP6q1bQYi+JsJ8jcfRNK04yetszR5+
ubxNTWf5HvJ0af+MJLNVqErXZVv82xlu+MI8AaHi0P2LpeyUbOn8bQsE+zMp8/I8
P/ZvrfR66VSul+GEVw5prtEcfS9OpkuL0kraVoGTCTlCJMsmfGKkf13YgdjRA+es
+ybof2Rn65PerSsXoUPkAOYIcVvAamNlx1A87m096vs6cNiRbPmduJiMyW+iGWm5
5V9jAoGBANikPWikNIz2HrbP9CVnU6VD2x4ikWlS3CZ18xNhSgafTww3JsrtTD3f
8h7jBVOtMutzMeVh7SwNke2y//paV0JXsl5TW0AoZXJhT6G/U2Eb3LqZF2nG9k5T
A+lockUOB/XT+jEBcdj5e7FhOYau199GPkFa4oZzQFnM7v56rMHDAoGBAMHMhsRt
+SFnQ+la7YWx3gPBj8t4vOSvaMllnNY6I0Dc2KSZJZLeSMQxK4C5muo1pkbiwEhe
V9SkLYFiumHqeYgPcoL70RcjPh/kWjdSr4qLDCm7bv5DQjMO3oeiYRrCHfLJMfxK
K44gZtvGhNkoQ6F4U7kTkcyxeWk/TdpD/3XrAoGBAKcfnRY1h5/xEGEkFBzarj2p
X+QMCcZHPB2rcl9oTwxsREtAfQLxkpYhyWhIuvdoWnseWbDzxZcDJ245oOWzZ+K4
```

```
riuCIdJoTkJLEYtntVmKhbSEukpJByZ6jf0zUNYvZTn7S/nrK/CF8bS4t1lFDO4R
nb1jXkp2KlPnPAYkHt6dAoGAQnICUfnRwk9Ke+SpNScKSm4/7f2Cl4yY9JfeAE5y
fKN9ekMi00Tlv5vUZqYJOluJd2Ovo8iTJFWL8di07l8FGLTWzdKuFo+9h0b6XBxZ
cVQ+rzjy2tU7RLSkET78uLeVPUxLpvTbu7NWdzy/1Wzj/NjMP3RsgLHge9swopr4
gDMCgYEAmrjRJUDBhb7AARJGIbbxneEmSVMxr0KX+dDGeD04+3obGg6Aasmbk9hP
t4/YlQLWBnxFqIVdjXBe/zRO5UgWQWTI85nxdyCaUOPBrITjmYggLGPo+ruYHp37
nmqZUago5RYGKJCgt/Ep0aZruagLG7tCF5T43Y8nnrLZ8slYoN0=
-----END RSA PRIVATE KEY-----
```

File "*sd_receipt_oem.key.pub*":

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEApADnOQYjBGDzdszfv1iP6L9k9m4d0a3xT/DvHT3kH7Uy7Tqu/K3uTNABvLthN4ZWmzR
yTUWxOeI5NHTYpRW1z4c8rn1VRFGOd+tArr+HW4+U5UhSC6P8Mvb9DolzJ1uPp6sqhe5LpQ/QnE8y5atCaA+Vz1rj63r/ZA
oFGrFRDkUgHcO7nbegJFEheb/st6S2Q1/9NAh6YZcXMZNg7IhOTZHcYhBPyk3FVvVGtLzNiTLyBL5YXvPH61tE5YHalAJSB
0J5yiaQtAOYmhtpjpAWLqgS4GrqLCGd2k84vcDkPfmVW0U88im1k0rlInqyCD1B0RrUTJ4Mzo3ZUsGEm2D9AQ== OEM RSA
public key for receipt log transfer
```

As you can see, the comment for this key was chosen to be "*OEM RSA public key for receipt log transfer*". This file (together with the RSA host public key) has to be transferred to G&D.

An administrator of G&D is now performing the following steps[9]:

1. The RSA host public key is added to "*/home/mobicore/.ssh/known_hosts*".

2. The file "*/home/mobicore/.ssh/authorized_keys*" is modified. A new line is added:

```
command="/bin/bash     -c     <path>/pull_receiptlogfile.sh",no-port-
forwarding,no-x11-forwarding,no-agent-forwarding                ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEApADnOQYjBGDzdszfv1iP6L9k9m4d0a3xT/DvHT3kH7U
y7Tqu/K3uTNABvLthN4ZWmzRyTUWxOeI5NHTYpRW1z4c8rn1VRFGOd+tArr+HW4+U5UhSC6
P8Mvb9DolzJ1uPp6sqhe5LpQ/QnE8y5atCaA+Vz1rj63r/ZAoFGrFRDkUgHcO7nbegJFEhe
b/st6S2Q1/9NAh6YZcXMZNg7IhOTZHcYhBPyk3FVvVGtLzNiTLyBL5YXvPH61tE5YHalAJS
B0J5yiaQtAOYmhtpjpAWLqgS4GrqLCGd2k84vcDkPfmVW0U88im1k0rlInqyCD1B0RrUTJ4
Mzo3ZUsGEm2D9AQ== OEM RSA public key for receipt log transfer
```

3. The shell script "*pull_receiptlogfile.sh*" is created. It is executed when an SSH connection is established from the remote (here: OEM) side. This script looks like this:

```
#!/bin/bash
LD_LIBRARY_PATH=<any additional paths>:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
RSYNC=<path-to-rsync-binary>/rsync
DSTDIR=/home/mobicore/<oem-specific-path>
${RSYNC} --rsync-path=${RSYNC} --server --delete . ${DSTDIR}/
<path>/set_db_import_trigger.sh
```

For each OEM, a dedicated path exists on the G&D server ("*/home/mobicore/<oem-specific-path>*"). The RSync command pulls all files from the remote server to this local directory. The

[9] The analogous steps have to be performed by an administrator of the OEM.

RSync command on the OEM server (see below) controls, which file(s) is/are transferred.

Please note that RSync requires a source and a destination directory, which are "*.*" (current directory) and "*${DSTDIR}*" in the example above. The source directory normally specifies a directory on the source (remote=OEM) host. This directory is ignored by the RSync/SSH combination. The OEM host controls, which path will be used (also for security reasons!).

After the OEM has generated a new receipt log file, a shell script has to be executed, which may look like this (sample!):

```
#!/bin/bash
export LD_LIBRARY_PATH=<any-additional-paths>:$LD_LIBRARY_PATH
RSYNC=<path-to-rsync-binary>/rsync
SSH=<path-to-ssh-binary>/ssh
SSHPORT=2222
SSHKEY=/home/mobicore/.ssh/sd_receipt_oem.key
SRCDIR=/home/mobicore/<receipt_log_file_dir>
DSTDIR=.
USER=mobicore
HOST=<ip-address of G&D host>

${RSYNC} --rsync-path=${RSYNC} --delete -e "${SSH} -p ${SSHPORT} ↵
-i ${SSHKEY}" ${SRCDIR}/*.txt ${USER}@${HOST}:${DSTDIR}
```

In this example, the OEM has created an arbitrary subdirectory "*<receipt_log_file_dir>*" under "*/home/mobicore*". This is the target location for the receipt log file generated by the Production Database. It is assumed that the file extension of the receipt log file is "*.txt*".

After the receipt log file was generated, the above listed shell script is executed. An RSync operation is initiated. The switch "-e" tells RSync to open an SSH connection to the specified remote host (G&D).

After the SSH connection has been established, the G&D communication partner executes the script "*pull_receiptlogfile.sh*" shown in item 3 on page 48. This opens the server side of the RSync connection. The server (G&D) is now waiting for the OEM machine.

The RSync on the OEM machine is now transferring all files from the source directory with the file extension "*.txt*" to the G&D server over SSH. After that, the connection is closed.

The final command "*set_db_import_trigger.sh*" in the script "*pull_receiptlogfile.sh*" triggers the processing of the receipt log file by the G&D Backend System.

After the receipt acknowledge file was generated, the reverse RSync/SSH operation is performed to let the OEM server pull the receipt acknowledge file.

The setup of this reverse transfer direction is not shown here. It is fully symmetric to the sample setup shown above.

3.5.1.4 Modifications of the receipt log and acknowledge file

The specification of the receipt log file (please refer to subsection 3.3.1 on page 40) and of the receipt acknowledge file (please refer to subsection 3.3.2 on page 40) are slightly modified for the RSync/SSH transfer mechanism.

Modification of the receipt log file

Deviating from subsection 3.3.1 on page 40, an additional text line SHALL precede the receipt log file.

The first line of the receipt log file SHALL contain the SHA-256 message digest and a transaction ID (TID) chosen by the OEM. The SHA-256 SHALL be computed over the entire receipt log file except for this very first line.

In compliance with subsection 3.5.2.5 on page 51, the first textual line SHALL be formatted as follows:

`TID:<`***`TID`***`>;MD:<`***`SHA256 message digest`***`>`

The data item *<SHA256 message digest>* shall consist of 64 hexadecimal digits containing the 32 bytes SHA-256 message digest of the receipt log file. Uppercase and lowercase letters are allowed (i.e. 'a'..'f' or 'A'..'F', respectively).

The TID SHALL be freely chosen by the OEM. It MUST NOT contain the semicolon ';'.

Modification of the receipt acknowledge file

Deviating from subsection 3.3.2 on page 40, the first line of the acknowledge file SHALL be modified as follows:

OLD: *<status code>;<status message>*
NEW: <TID>*;<status code>;<status message>*

The TID of the receipt log file sent by the OEM is repeated in the first line of the receipt acknowledge file. The OEM MAY use this TID to associate the acknowledge file with the receipt log file.

3.5.2 Fully-automated receipt transfer over E-mail

The G&D Backend System is able to receive E-mails from the OEM containing the receipt log file as a file attachment. In most of the cases, the receipt acknowledge file cannot be transferred back to OEM via E-mail. For this reason, a SOAP B2B interface exists, which can be used for this purpose (please refer to subsection 3.5.3 on page 52).

3.5.2.1 Preparation and format of the data

The device binding process outputs two information items for each device:

1. 128bit (16 bytes) SUID of the SoC;
2. 768 bytes SD.Receipt (cryptographically secured data object; RSA encrypted and digitally signed);

The OEM has to create a text-based "*receipt log file*" (please refer to subsection 3.3.1 on page 40).

At least, the above mentioned two data items have to be present in a line of the text file: the SUID and the BASE64-encoded SD.Receipt.

<u>Definition #1:</u>

The field separator (e.g. space, tabulator, comma, semicolon, etc.) can be freely chosen by the OEM but has to be communicated to G&D.

As defined in this document, the SUID **SHALL** consist of 32 hexadecimal digits and the SD.Receipt **SHALL** be BASE64-encoded (1024 characters).

3.5.2.2 Message digesting the receipt log file

The G&D Backend System shall have a way to check for the completeness of the receipt log file sent by the OEM.

For this purpose:

Definition #2

The OEM SHALL use the message digest SHA-256 (FIPS 180-2) to compute a hash of the text-based receipt log file[10].

3.5.2.3 Transaction ID

For the tracking of problems that might occur for a specific receipt log file transfer from the OEM to the G&D Backend System:

Definition #3:

The OEM SHALL assign a unique transaction identifier (TID) to each E-Mail sent to the G&D Backend System. The type and format of this TID SHALL be defined by the OEM. The TID MUST NOT contain a semicolon ';'.

3.5.2.4 Limitation of the data volume

Definition #4:

The OEM SHALL limit the amount of data sent to the G&D Backend System to 20 Megabytes.

3.5.2.5 E-Mail transfer to the G&D Backend System

G&D assigns a unique E-mail address to the OEM. The E-Mail address will look like:

<*oem*>@mcore.gi-de.com

Definition #5:

The subject of the E-Mails sent by the OEM to the G&D Backend System SHALL be formatted as:

TID:<*TID*>;MD:<*SHA256 message digest*>

---

[10] For the SHA256 message digest operation, the OpenSSL command line tool can be used.

The data item <*TID*> SHALL be the unique transaction identifier assigned by the OEM.

The data item <*SHA256 message digest*> shall consist of 64 hexadecimal digits containing the 32 bytes SHA-256 message digest of the receipt log file. The prefix "MD" stands for "*Message Digest*".

The mail body of the E-mail SHALL be empty. The E-mail SHALL contain one attachment, which is named "**receipts.log**" and contains the receipt log file (text-based).

Please note that the OEM and G&D have to exchange the IP addresses of the servers communicating with each other. The firewalls of both organizations shall be configured to allow traffic from/to the opposite side.

3.5.3 SOAP B2B interface (acknowledge transfer)

Please refer to appendix B on page 83. It shows the WSDL file declaring the SOAP B2B web interface, which can be used by the OEM as a template to implement the receipt acknowledge web service.

This web service is designed to acknowledge E-mail transfers from the OEM to G&D via the E-mail interface described in subsection 3.5.2 on page 50).

The G&D Backend Server collects any errors occurred during the database import of the receipts. Furthermore, it validates the message digest (SHA-256) sent by the OEM as part of the E-mail subject.

The OEM has to provide SSL/TLS X.509 certificates because the web service requires SSL server authentication. For authentication purposes, two methods exist:

1. HTTP basic authentication: The OEM has to provide a username and a password.
2. SSL/TLS client authentication: The OEM has to provide a PKCS#12 container containing an SSL client certificate and an RSA private key. G&D will use this information to perform an SSL client authentication.

It is up to the OEM to select one of these authentication mechanisms. Furthermore, both parties (OEM and G&D) shall exchange the IP addresses of the servers communicating with each other so that the firewalls of both enterprises can be set up properly.

A Appendix

A.1 Communication channel examples (USB)

A.1.1 Prerequisites

A USB connection between the Production Station and the Device requires the appropriate device drivers on the host side, e.g. for MS Windows, either a legacy driver or the generic WinUSB driver has to be installed.

If a **standard** Android image is deployed for the device binding on the Device, then the USB gadget driver shall already be available. USB bulk transfers are used to transfer data packets between the Production Station and the Device. The maximum packet size is 4095 bytes for USB 1.1 and 4096 bytes for USB 2.0. None of the device binding messages exceed this limit.

If a **non-standard** image (e.g. a temporary test image) is deployed for the device binding on the Device, then an appropriate USB driver has to be provided. It is recommended to use USB bulk transfers in this scenario, too.

A.1.2 Android Debug Bridge (ADB) and TCP/IP port forwarding

If the "*USB debugging*" feature can be temporarily enabled by the OEM for the device binding process, then a TCP/IP port can be easily forwarded from the Production Station to the Device to establish a bidirectional communication channel.

A.1.3 ADB infrastructure and USB bulk transfers

It is also possible to establish a USB connection from the Production Station (MS Windows: legacy or WinUSB driver) to the Device using the existing USB gadget driver of the Linux kernel.

An application running in the device can just open the file `/dev/android_adb_enable` for reading and writing. This signals the Linux kernel driver to activate the USB transport mechanism that is normally used by the ADB daemon. After that, the application opens the device file `/dev/android_adb` for reading and writing. This second file descriptor is used to read USB packets sent by the Production Station to the Device and to write USB packets to be sent by the Device to the Production Station. Root rights are required to access both device files.

The Production Station software can just use the MS Windows WinUSB API to communicate with the Device.

A.1.4 Direct USB connection

The OEM may also establish a direct USB connection between the Production Station and the Device. The technical details are outside the scope of this document.

It is just recommended to use the USB bulk transfer mode to transfer packets between the two communication partners.

A.2 Provisioning API C header file

```
///
/// @file       gdmcprovlib.h
/// @author     Giesecke & Devrient GmbH, Munich, Germany
///
/// This header file declares simple data types and functions
/// comprising the G&D Provisioning API.
///

#ifndef _INC_GDPROVLIB_H_
#define _INC_GDPROVLIB_H_

#include <stdlib.h>

#ifdef __cplusplus
extern "C" {
#endif

// Check defines (macros)...

#if !defined(WIN32) && !defined(LINUX) && !defined(ARM)
#error "You MUST define one of WIN32, LINUX, and ARM (platform)."
#endif

#if !defined(_32BIT) && !defined(_64BIT)
#error "You MUST define either _32BIT or _64BIT."
#endif

#if !defined(_LENDIAN) && !defined(_BENDIAN)
#error "You MUST define either _LENDIAN or _BENDIAN."
#endif

// Declare simple signed and unsigned integer types

/// a byte (octet), unsigned, 0..255
typedef unsigned char              _u8;

/// a signed byte, -128..+127
typedef signed char                _i8;

/// an unsigned 16bit integer, 0..65.535
typedef unsigned short             _u16;

/// a signed 16bit integer, -32.768..+32.767
typedef signed short               _i16;

/// an unsigned 32bit integer, 0..4.294.967.295
typedef unsigned int               _u32;

/// a signed 32bit integer, -2.147.483.648..+2.147.483.647
typedef signed int                 _i32;

#ifdef WIN32

#define GDPUBLIC
#define GDPROVAPI         __fastcall

/// an unsigned 64bit integer, 0..18.446.744.073.709.551.615
typedef unsigned __int64           _u64;
```

```
/// a signed 64bit integer, -9.223.372.036.854.775.808..+9.223.372.036.854.775.807
typedef signed __int64            _i64;

#else

#define GDPUBLIC          __attribute__((visibility("default")))
#define GDPROVAPI

#ifdef _32BIT

/// an unsigned 64bit integer, 0..18.446.744.073.709.551.615
typedef unsigned long long        _u64;

/// a signed 64bit integer, -9.223.372.036.854.775.808..+9.223.372.036.854.775.807
typedef signed long long          _i64;

#else // 64bit

/// an unsigned 64bit integer, 0..18.446.744.073.709.551.615
typedef unsigned long             _u64;

/// a signed 64bit integer, -9.223.372.036.854.775.808..+9.223.372.036.854.775.807
typedef signed long               _i64;

#endif // _32BIT

#endif // WIN32

/////////////////////////////////////////////////////////////////////////////

/// G&D error codes, which are unsigned 32bit integers
typedef _u32                          gderror;

/// everything okay, operation successful
#define GDERROR_OK                    ((gderror)0x00000000)

/// one or more of the input parameters to a function is/are invalid
#define GDERROR_PARAMETER             ((gderror)0x00000001)

/// connection problem occured, unable to establish a connection to the
/// Key Provisioning Host (KPH)
#define GDERROR_CONNECTION            ((gderror)0x00000002)

/// communication problem occured, unable to communicate with the
/// Key Provisioning Host (KPH)
#define GDERROR_COMMUNICATION         ((gderror)0x00000003)

/// GDMCProvShutdownLibrary was called without calling GDMCProvInitializeLibrary
#define GDERROR_NOT_INITIALIZED       ((gderror)0x00000004)

/// GDMCProvBeginProvisioning called but no more handles available
#define GDERROR_NO_MORE_HANDLES       ((gderror)0x00000005)

/// An unknown or invalid gdhandle was passed to a function
#define GDERROR_INVALID_HANDLE        ((gderror)0x00000006)

/// A so called structured exception occured, which is a severe error
/// (MS Windows only)
#define GDERROR_CPU_EXCEPTION         ((gderror)0x00000007)

/// Unable to retrieve the SUID of the SoC
#define GDERROR_CANT_GET_SUID         ((gderror)0x00000008)

/// Unable to generate the authentication token SO.AuthToken
```

```
#define GDERROR_CANT_BUILD_AUTHTOKEN   ((gderror)0x00000009)

/// Unable to dump the authentication token SO.AuthToken
#define GDERROR_CANT_DUMP_AUTHTOKEN    ((gderror)0x0000000A)

/// Unable to generate the receipt SD.Receipt
#define GDERROR_CANT_BUILD_RECEIPT     ((gderror)0x0000000B)

///  (only  product  version):  Authentication  KPH  Connector  <->  Key  Provisioning  Host
(KPH) failed
#define GDERROR_AUTH_FAILED            ((gderror)0x0000000C)

/// validation of the device binding failed
#define GDERROR_VALIDATION_FAILURE     ((gderror)0x0000000D)

/// insufficient memory available
#define GDERROR_INSUFFICIENT_MEMORY    ((gderror)0x0000000E)

/// synchronization error occurred (thread concurrency)
#define GDERROR_SYNCHRONIZATION        ((gderror)0x0000000F)

/// the Key Provisioning Host (KPH) was not able to generate a random key (TRNG)
#define GDERROR_CANT_GENERATE_KEY      ((gderror)0x00000010)

/// the received cryptographic message format is erroneous
#define GDERROR_MESSAGE_FORMAT         ((gderror)0x00000011)

/// CRC32 checksum error
#define GDERROR_CRC32                  ((gderror)0x00000012)

/// Hash value (message digest) validation error
#define GDERROR_MESSAGE_DIGEST         ((gderror)0x00000013)

/// SUID comparison failed
#define GDERROR_SUID_MISMATCH          ((gderror)0x00000014)

/// the Device could not generate the authentication token SO.AuthToken for any reason
#define GDERROR_GENAUTHTOK_FAILED      ((gderror)0x00000015)

/// the Device could not wrap the authentication token in a secure object (SO)
#define GDERROR_WRAPOBJECT_FAILED      ((gderror)0x00000016)

/// the Device could not store SO.AuthToken for any reason
#define GDERROR_STORE_SO_FAILED        ((gderror)0x00000017)

///  the  Key  Provisioning  Host  (KPH)  could  not  generate  the  receipt  SD.Receipt  for  any
reason
#define GDERROR_GENRECEIPT_FAILED      ((gderror)0x00000018)

///  the  Key  Provisioning  Host  (KPH)  triggered  a  SO.AuthToken  validation  in  the  Device
but no SO.AuthToken is available
#define GDERROR_NO_AUTHTOK_AVAILABLE   ((gderror)0x00000019)

/// the Device could not perform a read-back of the recently stored SO.AuthToken
#define GDERROR_AUTHTOK_RB_FAILED      ((gderror)0x0000001A)

/// the called API function is not implemented
#define GDERROR_NOT_IMPLEMENTED        ((gderror)0x0000001B)

/// generic (unspecified) error
#define GDERROR_UNKNOWN                ((gderror)0x0000001C)

/// MobiCore library initialization or cleanup failed
#define GDERROR_MOBICORE_LIBRARY       ((gderror)0x0000001D)
```

```
/// supplied (output) buffer too small
#define GDERROR_BUFFER_TOO_SMALL      ((gderror)0x0000001E)

/// cryptographic-related error occured, e.g. loading of RSA keys, etc.
#define GDERROR_CRYPTO_FAILURE        ((gderror)0x0000001F)

/// the API function GDMCProvSetConfigurationString failed to parse
/// the configuration string
#define GDERROR_CONFIGURATION_ERROR   ((gderror)0x00000020)

/// no error code: device binding completed successfully
#define GDERROR_PROVISIONING_DONE     ((gderror)0x10000001)

//////////////////////////////////////////////////////////////////////////////

/// G&D handle (to one instance of the Provisioning API)
typedef _u32                      gdhandle;

/// Returns the current version of the Provisioning API.
///
/// @return an unsigned 32bit integer consisting of four bytes aa|bb|cc|dd
///         with major version (aa), minor version (bb), patch level (cc), and
///         OEM (dd), which denotes the numeric ID of an OEM.
GDPUBLIC _u32 GDPROVAPI GDMCProvGetVersion ( void );

/// [PRODUCTION STATION ONLY] Formats an error message for an error code,
/// possibly containing more detailed information about the error. This function
/// is NOT implemented in the ARM version of the library because no diagnostic
/// messages can be displayed during the production.
///
/// @param[in]      provhandle  the handle returned by GDMCProvBeginProvisioning;
///                             can be null (0) to format a message for a global
///                             error code (not context-specific)
/// @param[in]      errorcode   the G&D error code
/// @param[in/out]  msgbuf      pointer to buffer receiving the UTF-8 encoded
///                             error message (in), buffer filled with error
///                             message (out)
/// @param[in/out]  size        size of buffer pointed to by msgbuf specified
///                             as wide characters (in), number of wide
///                             characters copied into msgbuf (out)
///
/// @return                     result code (e.g. buffer too small)
GDPUBLIC gderror GDPROVAPI GDMCProvFormatErrorMessage ( gdhandle provhandle,
                                                        gderror  errorcode,
                                                        char     *msgbuf,
                                                        _u32     *size );

/// Initializes the G&D Provisioning API (library) globally. If called
/// by the Production Software Station, then a TLS-secured channel to
/// the Key Provisioning Host (KPH) is established.
/// In a multithreaded environment, this function has to be called from
/// the primary thread (LWP 0).
///
/// @return G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvInitializeLibrary ( void );

/// Performs a global shutdown of the G&D Provisioning API (library).
/// After this call, all resources are cleaned up and all handles are
/// closed. No functions except for GDMCProvInitializeLibrary may be
/// called anymore.
/// In a multithread environment, this function has to be called from
/// the primary thread (LWP 0).
///
```

```
/// @return G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvShutdownLibrary ( void );

/// Creates one instance of the key provisioning (aka "device binding")
///
/// @param[in/out]  provhandle  pointer to memory location receiving the
///                             handle (in), the handle or 0 (out)
///
/// @return                     G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvBeginProvisioning ( gdhandle *provhandle );

/// Destroys one instance of the key provisioning (aka "device binding")
///
/// @param[in]  provhandle      the handle returned by GDMCProvBeginProvisioning
///
/// @return                     G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvEndProvisioning ( gdhandle provhandle );

/// Executes one provisioning step of the full sequence. The caller has to
/// call this function in a loop until either an error is reported or the
/// error code GDERROR_PROVISIONING_DONE is returned (meaning successful
/// provisioning). Please refer to the MobiCore Provisioning API documentation
/// for details.
///
/// @param[in]      provhandle      the handle returned by
///                                 GDMCProvBeginProvisioning
/// @param[in]      msgin           pointer to buffer containing the
///                                 input message; may be NULL if no message
///                                 available
/// @param[in]      msgin_size      size of buffer pointed to by msgin in bytes
/// @param[in/out]  msgout          pointer to buffer receiving the output
///                                 message (in); output message (out)
/// @param[in/out]  msgout_size     size of buffer pointed to by msgout in
///                                 bytes (in); number of bytes copied to msgout
///                                 (out)
///
/// @return                         G&D error code; GDERROR_PROVISIONING_DONE
///                                 if provisioning successfully completed.
GDPUBLIC gderror GDPROVAPI GDMCProvExecuteProvisioningStep (
                  gdhandle    provhandle,
                  const _u8   *msgin,
                  _u32         msgin_size,
                  _u8         *msgout,
                  _u32        *msgout_size );

/// [PRODUCTION STATION ONLY] Convenience function to format an SD.Receipt
///
/// @param[in]      receipt           pointer to buffer containing the
///                                   binary SD.Receipt
/// @param[in]      receipt_size      size of binary data pointed to by
///                                   receipt in bytes
/// @param[in/out]  fmt_receipt       pointer to buffer receiving the receipt as
///                                   a BASE64-encoded string (in); the string (out)
/// @param[in/out]  fmt_receipt_size  size of buffer pointed to by fmt_receipt in
///                                   bytes (in); number of bytes copied to
///                                   fmt_receipt (out)
///
/// @return                           G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvFormatReceipt (
                  const _u8   *receipt,
                  _u32         receipt_size,
                  _u8         *fmt_receipt,
                  _u32        *fmt_receipt_size );
```

```
/// [PRODUCTION STATION ONLY] Convenience function to query the SUID of
/// the currently provisioned device (e.g. can be used as primary key in
/// a production database)
///
/// @param[in]      provhandle    the handle returned by
///                               GDMCProvBeginProvisioning
/// @param[in/out]  suid          pointer to buffer (16 octets, in) receiving the
///                               SUID of the current mobile device (out)
///
/// @return                       G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvGetSUID (
                  gdhandle    provhandle,
                  _u8        *suid );

/// [DEVICE ONLY] Callback function called by the Provisioning API when
/// GDMCProvExecuteProvisioningStep is executed in the Device. This function
/// shall store the authentication token SO.AuthToken in a secure location.
///
/// @param[in]      authtok           pointer to buffer containing SO.AuthToken
/// @param[in]      authtok_size      size of buffer pointed to be authtok;
///                                   shall be 148 octets
///
/// @return                           G&D error code
typedef gderror (*authtok_writecb)( const _u8 *authtok,
                                    _u32       authtok_size );

/// [DEVICE ONLY] Callback function called by the Provisioning API when
/// GDMCProvExecuteValidationStep is executed in the Device. This function
/// shall perform a read-back of the stored authentication token SO.AuthToken
///
/// @param[in/out]  authtok           pointer to buffer receiving SO.AuthToken
///                                   (in); buffer filled with SO.AuthToken (out)
/// @param[in/out]  authtok_size      size of buffer pointed to be authtok (in);
///                                   number of bytes copied to authtok (out);
///                                   shall be 148 octets
///
/// @return                           G&D error code
typedef gderror (*authtok_readcb)( _u8  *authtok,
                                   _u32 *authtok_size );

/// [DEVICE ONLY] The OEM must provide two hook functions (callbacks) for the
/// reading and writing of the authentication token SO.AuthToken in the device.
///
/// @param[in]  writefunc   callback function called by the Provisioning API
///                         when an authentication token SO.AuthToken has to be
///                         stored
/// @param[in]  readfunc    callback function called by the Provisioning API
///                         when an authentication token SO.AuthToken has to be
///                         read back (for validation purposes)
///
/// @return                 G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvSetAuthTokenCallbacks (
                            authtok_writecb writefunc,
                            authtok_readcb  readfunc );

/// [PRODUCTION STATION ONLY] The configuration of the provisioning library
/// can be patched into the library binary file. If the OEM decided to perform
/// the configuration e.g. by providing the configuration information via the
/// production database, then this function can be called to configure the
/// provisioning library.
///
/// @param[in] config_string  a zero-terminated configuration string containing
///                           the entire configuration information in a format
///                           that will be defined by G&D; the exact format of
```

```
///                          this configuration information can be OEM-specific
///                          and will be specified in a separate document
///
/// @return                  G&D error code
GDPUBLIC gderror GDPROVAPI GDMCProvSetConfigurationString (
                            const char *config_string );

//////////////////////////////////////////////////////////////////////////////
// Declaration of message header and trailer
//////////////////////////////////////////////////////////////////////////////

#ifdef WIN32

#pragma warning ( disable : 4200 )

#pragma pack(push,1)

#define PACK_ATTR

#else // Linux

#define PACK_ATTR   __attribute__((packed))

#endif

typedef struct _gdmc_msgheader          gdmc_msgheader;
typedef struct _gdmc_msgtrailer         gdmc_msgtrailer;

/// the G&D MobiCore message header
struct _gdmc_msgheader
{
  _u32        msg_type;   ///< message type
  _u32        body_size;  ///< size of body (may be 0)
} PACK_ATTR;

/// the G&D MobiCore message trailer
struct _gdmc_msgtrailer
{
  _u32        magic;      /// message type (one's complement)
  _u32        crc32;      /// CRC32 checksum
} PACK_ATTR;

#ifdef WIN32
#pragma pack(pop)
#endif

#ifdef __cplusplus
}
#endif

#endif // _INC_GDPROVLIB_H_
```

A.3 The cryptographic protocol (normative)

The device binding mainly relies on the two messages:

- GenerateAuthToken (subsection 10.1.1 in **Error! Reference source not found.**)
- GenerateReceipt (subsection 10.1.2 in **Error! Reference source not found.**)

The full cryptographic protocol requires the definition of a more detailed protocol, which honors the design goals:

1. fault tolerant
2. capable of handling errors
3. lightweight
4. complete (w.r.t. all requires exchanges)
5. capable of detecting transmission errors (CRC)

The following subsections detail the messages consisting of a message header followed by a message body followed by a message trailer.

A.3.1 Message format

The multi-byte numeric values are stored in Little Endian format (according to the ADB protocol).

Table 3 shows the generic message format:

Value:	Type:	Length:	Description:
MESSAGE HEADER			
message type	uint32	4	type of message (numeric constant)
size of body	uint32	4	length of message following this value (in octets)
MESSAGE BODY			
message body	uint8	variable	the message body (may be empty)
MESSAGE TRAILER			
magic	uint32	4	one's complement of message type (see above)
crc32	uint32	4	CRC32 checksum (Cyclic Redundancy Check)

Table 3: Cryptographic message format.

The message overhead is 16 octets (bytes) per message.

A.3.2 Messages

A.3.2.1 GetSUID request

The message *GetSUID request* is sent from the Production Station to the Device to request the SUID of the SoC.

The message type is `MC_GETSUID_REQ`.

The message body is empty.

The overall size of this message is **16** octets (bytes) including header and trailer.

A.3.2.2 GetSUID response

The message *GetSUID response* is sent from the Device to the Production Station in response to the *GetSUID request* message.

The message type is `MC_GETSUID_RESP`.

The message body contains the **16** octets (bytes) representing the SUID of the SoC.

The overall size of this message is **32** octets (bytes) including header and trailer.

A.3.2.3 GenerateAuthToken request

The message *GenerateAuthToken request* is sent from the Production Station to the Device requesting the Device to generate the authentication token SO.AuthToken.

The message type is `MC_GENAUTHTOKEN_REQ`.

The message body is the *GenerateAuthTokenMsg* (*ActMsg*) as defined in subsection 10.1.1 in **Error! Reference source not found.**:

Offset:	Type:	Length:	Value:
0	uint32	4	MC_CMP_CMD_GENERATE_AUTH_TOKEN
4	uint8[16]	16	SUID (sequence of octets)
20	uint8[32]	32	K.SoC.Auth (generated by TRNG of Key Provisioning Host)
52	uint32	4	KID (Key Identifier)
56	uint8[256]	256	PKCS#1 PSS RSA signature over all preceding fields (2048bit RSA key)

Table 4: GenerateAuthTokenMsg (message body of GenerateAuthToken request).

The size of the message body is **312** octets (bytes).

The overall size of this message is **328** octets (bytes) including header and trailer.

A.3.2.4 GenerateAuthToken response

The message *GenerateAuthToken response* is sent from the Device to the Production Station in response to the *GenerateAuthToken request* message.

The message type is `MC_GENAUTHTOKEN_RESP`.

The message body is (according to subsection 10.1.1 in **Error! Reference source not found.**):

Offset:	Type:	Length:	Value:
0	uint32	4	MC_CMP_CMD_GENERATE_AUTH_TOKEN_RSP
4	uint32	4	result code
8	uint8[152]	152	Secure Object SO.AuthToken

Table 5: GenerateAuthTokenMsg (command response).

The size of the message body is **160** octets (bytes).

The authentication token SO.AuthToken (152 octets) is comprised of the following data items:

Offset:	Type:	Length:	Value:
			HEADER
0	uint32	4	tpe
4	uint32	4	version
8	uint32	4	context
12	uint32	4	lfetime
16	uint32	4	producer spid
20	uint8[16]	16	producer uuid
36	uint32	4	plain_length (28)
40	uint32	4	encrypted_length (32)
			PLAINTEXT DATA
44	uint32	4	Content type
48	uint32	4	Content version
52	uint32	4	Content state
56	uint8[16]	16	SUID (sequence of octets)
			ENCRYPTED DATA (K.Device.Ctxt)
72	uint8[32]	32	K.SoC.Auth
104	uint8[32]	32	Hash (SHA256)
136	uint8[16]	16	ISO padding (0x80,0x00,…,0x00)

Table 6: Secure object SO.AuthToken.

The encryption is performed using AES-256-CBC with the standard block size of 128 bits (16 octets).

The SHA256 message digest is computed over the header, the plaintext data, and the K.SoC.Auth (plain).

The overall size of this message is **176** octets (bytes) including header and trailer.

A.3.2.5 ValidateAuthToken request

The message *ValidateAuthToken request* is sent from the Production Station to the Device to trigger a read-back of the recently generated authentication token SO.AuthToken.

The message type is `MC_VALIDATEAUTHTOKEN_REQ`.

The message body contains the SO.AuthToken (**152** octets) as received by the Production Station.

The overall size of this message is **168** octets (bytes) including header and trailer.

There is no *ValidateAuthToken response* message defined in this document. The Device answers with an error message (please refer to subsection A.3.2.6) in response to the *ValidateAuthToken request* message – possibly signaling success (`GDERROR_PROVISIONING_DONE`).

A.3.2.6 Error message

Because all message exchanges may result in errors, a dedicated error message is defined as follows:

The message type is `MC_ERROR`.

The message body is shown in Table 7.

Offset:	Type:	Length:	Value:
0	uint32	4	error code
4	uint32	4	error message length (optional, may be 0)
8	uint8[?]	var.	optional error message (UTF-8 encoded)

Table 7: Error message body.

A.3.3 Message exchanges and possible responses

Table 8 details the message exchanges for the use cases "*device binding*" and "*device binding validation*".

For each (request) message, all possible response messages are listed. The handling of all other error situations is performed by the internal DFA (***D****eterministic* ***F****inite* ***A****utomata*) implemented by the Provisioning API libraries, e.g. lost packets, network timeouts, etc.

Sender:	**Message:**	**Possible responses (follow-up message):**
	DEVICE BINDING MESSAGE EXCHANGE	
Prod.Station	MC_GETSUID_REQ	MC_GETSUID_RESP MC_ERROR(GDERROR_MESSAGE_FORMAT) MC_ERROR(GDERROR_CRC32) MC_ERROR(GDERROR_CANT_RETRIEVE_SUID)
Device	MC_GETSUID_RESP	MC_GENAUTHTOKEN_REQ
Prod.Station	MC_GENAUTHTOKEN_REQ	MC_GENAUTHTOKEN_RESP MC_ERROR(GDERROR_MESSAGE_FORMAT) MC_ERROR(GDERROR_CRC32) MC_ERROR(GDERROR_MESSAGE_DIGEST) MC_ERROR(GDERROR_SUID_MISMATCH) MC_ERROR(GDERROR_GENAUTHTOK_FAILED) MC_ERROR(GDERROR_WRAPOBJECT_FAILED) MC_ERROR(GDERROR_STORE_SO_FAILED)
Device	MC_GENAUTHTOKEN_RESP	MC_VALIDATEAUTHTOKEN_REQ
Prod.Station	MC_VALIDATEAUTHTOKEN_REQ	MC_ERROR(GDERROR_PROVISIONING_DONE) MC_ERROR(GDERROR_NO_AUTHTOK_AVAILABLE) MC_ERROR(GDERROR_AUTHTOK_RB_FAILED) MC_ERROR(GDERROR_VALIDATION_FAILURE) MC_ERROR(GDERROR_CRC32)

Table 8: Message exchanges and possible responses.

A.4 MobiCore® Key Provisioning SDK

The current version of this document is accompanied by a full implementation of the Provisioning API for MS Windows as well as for Android (ARM platform).

The SDK CD-ROM contains source codes, binaries, Visual Studio 2008 project files, and Android .mk makefiles.

The OEM may use this package to begin the integration of the Provisioning API in the Production Software. All cryptographic operations are performed by OpenSSL 1.0.0, **no MobiCore®-enabled device is necessary** to execute the binaries of this SDK package.

A.4.1 Contents of the SDK package (CD-ROM):

Figure 10 illustrates the folder structure of the SDK:

Figure 10: Folder structure of the SDK.

A.4.1.1 The folder "*AllPlatforms*"

This folder just contains a subfolder called "*inc*" that contains the C header file "*gdmcprovlib.h*", which is valid for both MS Windows and Android (Linux).

A.4.1.2 The folder "*Android*"

This folder contains sources codes and binaries for the Android platform.

The subfolder "*lib*" contains the shared objects (in the two subfolders "*Debug*" and "*Release*").

The subfolder "*MCKeyProvisioningSDK*" contains the subfolder "*DevBindingTest*" with the source code and Android.mk file to build the test tool "*DevBindingTest*" on Android. A "*README.txt*" file with build instructions is included, too.

The last subfolder, "*openssl*", contains the shared object "*libcrypto.so*" and the OpenSSL header files. This subfolder is required to build the "*DevBindingTest*" tool with the Android NDK. Please do not install the shared object on a target device because the Android platform comes with its own build – this one is required only by the link step.

A.4.1.3 The folder "*doc*"

… contains the document in-hand.

A.4.1.4 The folder "*Windows*"

This folder contains MS Windows executables and an MS Visual Studio 2008 solution to build the MS Windows test tools from scratch.

The subfolder "*bin*" is subdivided into the two subfolders "*Debug*" and "*Release*". It contains the prebuilt binaries and libraries (DLLs). The post-build steps of the included Visual Studio solution (see below) copy executable files into these folders.

Important note:
All DLLs in the subfolder Windows/bin **emulate** a Key Provisioning Host. If you want to use the production version (i.e. the KPH connector), then you have to use the DLL **GDMCProvLib32.dll from the subfolder *Windows/KPHConnector***.

The subfolder "*lib*" contains import libraries (Debug and Release).

Please note that the DLL "*GDMCProvLib32.dll*" ("*GDMCProvLib32d.dll*" – debug version) is the Key Provisioning DLL for MS Windows. The other library "*GDMCProvLibARM32(d).dll*" is included to support an MS Windows-only test run of the tools. It is the ARM version of the library compiled for MS Windows.

The subfolder "*MCKeyProvisioningSDK*" contains the MS Visual Studio solution with the two projects "*AndroidDevMockUp*" and "*ProductionStationMockUp*".

A new folder was recently added to the folder "*MCKeyProvisioningSDK*", which is named "*MCKeyProvisioningSDK**V6***". This folder contains Microsoft Visual Studio 6 versions of the workspaces (.dsw/.dsp files instead of .sln/.vcproj files). This is a supplement for OEMs that do not use Microsoft Visual Studio 2008.

A.4.2 Technical background

The cryptographic protocol is transferred by this demonstration software over TCP/IP. If you open two command line prompts, then please execute `AndroidDevMockUp.exe` first. It acts like a TCP/IP server listening on localhost (127.0.0.1), port 12000. The only recognized command line paramter is the string "*<ip>*:*<port>*", which can be used to switch to an alternative IP address and/or port.

If you now execute `ProdStationMockUp.exe` in the second command line prompt, then the key provisioning (device binding) is performed between these two communication partners on one single MS Windows host.

Alternatively, you can build the `DevBindingTest` (from the Android SDK folder) as a Linux native executable targeted to an Android device. A suitable `Android.mk` file is part of the SDK. This executable acts as a real Android implementation and can communicate over TCP/IP with the `ProdStationMockUp.exe`. The `AndroidDevMockUp.exe` is not required in this case – this part is handled by `DevBindingTest`.

Please refer to appendix A.4.3 on page 70 for the complete instructions.

A.4.3 Compiling and linking

A.4.3.1 MS Windows

Just open the top-level Visual Studio 2008 solution file and perform a batch rebuild. Alternatively, you can use the Visual Studio 6 files.

All executable files (DLLs and EXEs) are statically linked, i.e. you do not need the dynamic C Runtime on the target machine.

A.4.3.2 Android / Linux

You need the latest Android NDK (Native Development Kit) installed on your machine. The Android makefile `Android.mk` (located in the subfolder `Android/MCKeyProvisioningSDK/DevBindingTest/jni`) looks like this:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE       := DevBindingTest

LOCAL_C_INCLUDES   := $(LOCAL_PATH)/../inc \
                      $(LOCAL_PATH)/../../../../AllPlatforms/inc

LOCAL_SRC_FILES    := ../src/main.c \
                      ../src/tcpipnetworking.c
```

```
LOCAL_CFLAGS      := -O2 -Wall -fomit-frame-pointer -D$(MC_DEBUG) \
                     -DANDROID_ARM -DARM -D_LENDIAN -D_32BIT \
                     -fvisibility=hidden \
                     -I$(LOCAL_PATH)/../../../../Android/openssl/inc

LOCAL_LDFLAGS     := -Wl,-rpath-\
link,$(LOCAL_PATH)/../../../../Android/openssl/lib \
                     -Wl,-rpath-\
link,$(LOCAL_PATH)/../../../../Android/lib/$(MC_LIB_VER) \
                     -L$(LOCAL_PATH)/../../../../Android/openssl/lib \
                     -L$(LOCAL_PATH)/../../../../Android/lib/$(MC_LIB_VER) \
                     -lcrypto -lgdmcprov

include $(BUILD_EXECUTABLE)
```

Please read the file "*README.txt*" for detailed build instructions.

A.4.4 Executing the mock-ups

Please activate the option "*USB debugging*" on your Android device (this effectively starts the ADB daemon). Connect the device to your Windows host via USB.

On your Windows host, please start the ADB daemon by entering:

`adb forward tcp:12000 tcp:12000`

This will start the ADB daemon on the Windows machine as well as establish the required TCP/IP port forwarding (local: 12000, remote: 12000).

Open an ADB shell:

`adb shell`

Then, please change the current working directory to the folder in which you installed the binaries (please refer to the preceding subsection).

Adjust the environment variable `LD_LIBRARY_PATH` to include your current working directory. Please always use absolute paths here (for security reasons).

You can now execute the ARM binary by entering:

`./DevBindingTest` **OR:** `./DevBindingTestDebug`

The following messages will appear:

```
Initializing GDMCProvLibrary...OK.
Acquiring provisioning handle...OK.
Starting TCP/IP server and waiting for incoming data...
```

Please open a second console window on your Windows host. Please enter:

`ProdStationMockUp`

The device binding (key provisioning) protocol is now performed.

The test tools dump the following files in the current folder:

- SD.Receipt (768 bytes binary file containing the SD.Receipt)
- SUID.txt (dump of the SUID)

- receipt.log (SUID as 32 hexadecimal digits followed by a pipe symbol '|’ followed by the SD.Receipt BASE64-encoded)

A.4.5 Source code of main functions (informative)

A.4.5.1 Main function (ProdStationMockUp)

```
///
/// @file       main.c
/// @author     Giesecke & Devrient GmbH, Munich, Germany
///
/// mock-up implementation of the Production Station initiating
/// the device binding to one Android device.
/// This source code can be compiled for MS Windows.
/// The Windows version is suitable for either local (using
/// the Windows build of the AndroidDevMockUp) or remote
/// testing over the Android Debug Bridge with activated TCP
/// forwarding.
/// The Android device does not have to be MobiCore-enabled
/// because all cryptographic operations are simulated
/// by OpenSSL.

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#define SUID_SIZE          16        // 16 octets SUID
#define RECEIPT_SIZE       768       // 768 octets (three times 256 octets = 2048 bits)
#define MAX_PROV_TRIES      5            // perform five consecutive provisioning tries
(before giving up)

#pragma warning ( disable : 4996 )

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>

#include <gdmcprovlib.h>
#include <tcpipnetworking.h>

/// This is the main profisioning function.
///
/// @return process exit code
static int PerformProvisioning ( void )
{
  gderror                 err;
  gdhandle                provhandle;
  static _u8              msgin[MAX_BUFFER_SIZE];
  static _u8              msgout[MAX_BUFFER_SIZE];
  _u32                    msgin_size;
  _u32                    msgout_size;
  _u8                     suid[SUID_SIZE];
  _u8                     b64_receipt[1536];
  _u32                    b64_receipt_size;
  FILE                   *f;

  fprintf(stdout,"Perfoming key provisioning (device binding).\n");

  // 1.) Acquire handle

  err = GDMCProvBeginProvisioning(&provhandle);
  if (GDERROR_OK!=err)
  {
```

Please note that the modifier "**static**" **MUST NOT BE USED** in a multi-threaded environment. It is just used here for illustration purposes in the single-threaded Production Station mock-up.

```
    fprintf(stderr,"ERROR: GDMCProvBeginProvisioning failed.\n");
    return 1;
  }

  // 2.) Perform provisioning loop

  msgin_size  = 0; // signal no previous message available
  msgout_size = sizeof(msgout); // initialize with available bytes

  fprintf(stdout,"Entering provisioning loop.\n");

  while (GDERROR_OK==(err=GDMCProvExecuteProvisioningStep(
         provhandle,msgin,msgin_size,msgout,&msgout_size)))
  {
    // send message to device (if available)

    if (0!=msgout_size)
    {
#ifdef _DEBUG
      fprintf(stdout,"SEND TO DEVICE: %u byte(s):\n",msgout_size);
      gdmc_hexdump(msgout,msgout_size);
#endif
      if (!comm_send(msgout,msgout_size))
      {
        fprintf(stderr,"ERROR: send to device failed.\n");
        err = GDERROR_UNKNOWN; // use unknown error code to signal transmissione error
        break;
      }
    }

    // receive next message from device

    msgin_size = sizeof(msgin);
    if (!comm_recv(msgin,&msgin_size))
    {
      fprintf(stderr,"ERROR: recv from device failed.\n");
      err = GDERROR_UNKNOWN; // use unknown error code to signal transmissione error
      break;
    }
#ifdef _DEBUG
    fprintf(stdout,"RECV FROM DEVICE: %u byte(s):\n",msgin_size);
    gdmc_hexdump(msgin,msgin_size);
#endif

    // Check if we have to abort the provisioning loop

    if (GDERROR_OK!=err)
      break;

    msgout_size = sizeof(msgout); // initialize with available bytes (for next
iteration)
  }

  // check if device binding successful or error occured

  switch(err)
  {
    case GDERROR_PROVISIONING_DONE:
      fprintf(stdout,"Provisioning and validation successfully performed.\n");
      break;

    case GDERROR_VALIDATION_FAILURE:
      fprintf(stderr,"ERROR: Provisioning performed but validation FAILED.\n");
      GDMCProvEndProvisioning(provhandle);
```

```
      return 1;

    default:
      fprintf(stderr,"ERROR: Provisioning not performed due to errors.\n");
      GDMCProvEndProvisioning(provhandle);
      return 1;
  }

  // 3.) Get SUID and SD.Receipt for this device to store it in the production
database

  err = GDMCProvGetSUID(provhandle,suid);
  if (GDERROR_OK!=err)
  {
    fprintf(stderr,"ERROR: Unable to get the SUID of the SoC. ABORT.\n");
    GDMCProvEndProvisioning(provhandle);
    return 1;
  }

  f = fopen("SD.Receipt","wb");
  if (NULL!=f)
  {
    fwrite(msgout,1,msgout_size,f);
    fclose(f);
  }

  f = fopen("SUID.txt","wt");
  if (NULL!=f)
  {
    fprintf(f,"%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X",
      suid[ 0],suid[ 1],suid[ 2],suid[ 3],
      suid[ 4],suid[ 5],suid[ 6],suid[ 7],
      suid[ 8],suid[ 9],suid[10],suid[11],
      suid[12],suid[13],suid[14],suid[15]);
    fclose(f);
  }

  b64_receipt_size = sizeof(b64_receipt);
  memset(b64_receipt,0,sizeof(b64_receipt_size));

  if
(GDERROR_OK!=GDMCProvFormatReceipt(msgout,msgout_size,b64_receipt,&b64_receipt_size))
    fprintf(stderr,"ERROR: Unable to format SD.receipt.\n");
  else
  {
    fprintf(stdout,"B64 receipt size is %u\n",b64_receipt_size);
    b64_receipt[b64_receipt_size] = '\0';
    f = fopen("receipt.log","wt");
    if (NULL!=f)
    {

fprintf(f,"%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X|%s\n",
        suid[ 0],suid[ 1],suid[ 2],suid[ 3],
        suid[ 4],suid[ 5],suid[ 6],suid[ 7],
        suid[ 8],suid[ 9],suid[10],suid[11],
        suid[12],suid[13],suid[14],suid[15],b64_receipt);
      fclose(f);
    }
  }

#ifdef _DEBUG

  fprintf(stdout,"\nSUID    of    SoC:    %02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-
%02X%02X%02X%02X%02X%02X\n",
```

```
    suid[ 0],suid[ 1],suid[ 2],suid[ 3],
    suid[ 4],suid[ 5],suid[ 6],suid[ 7],
    suid[ 8],suid[ 9],suid[10],suid[11],
    suid[12],suid[13],suid[14],suid[15]);

  fprintf(stdout,"\nSD.Receipt:\n");

  gdmc_hexdump(msgout,msgout_size);

#endif

  return 0;
}

int main ( int argc, char *argv[] )
{
  gderror                  err;
  int                      exitcode = 0;

  // 0.) Override IP and port if specified

  if (argc>1)
    comm_setip_port(argv[1]);

  // If second parameter specified, then it is assumed to be a KPH
  // configuration string

  if (argc>2)
    GDMCProvSetConfigurationString(argv[2]);

  // 1.) Initialize GDMCProvLib

  fprintf(stdout,"Initializing GDMCProvLibrary...");
  fflush(stdout);

  err=GDMCProvInitializeLibrary();
  if (GDERROR_OK!=err)
  {
    fprintf(stderr,"ERROR.\n");
    return 1;
  }
  fprintf(stdout,"OK.\n");

  // 2.) Initialize TCP/IP networking

  fprintf(stdout,"Opening TCP/IP connection to device...");
  fflush(stdout);

  if (!comm_setup(0))
  {
    fprintf(stderr,"ERROR.\n");
    return 1;
  }

  // 3.) Perform the provisioning loop

  exitcode = PerformProvisioning();

  // 4.) Perform cleanup...

  comm_cleanup();

  err = GDMCProvShutdownLibrary();
```

```
  if (GDERROR_OK!=err)
  {
    fprintf(stderr,"ERROR: shutdown of provisioning library failed.\n");
    exitcode = 1;
  }

  return exitcode;
}
```

A.4.5.2 Main function (AndroidDevMockUp)

```
///
/// @file       main.c
/// @author     Giesecke & Devrient GmbH, Munich, Germany
///
/// mock-up implementation of an Android native executable
/// handling the MobiCore Device Binding messages originating
/// from the Production Station (mock-up).
/// This source code can be compiled for MS Windows or
/// Android (Froyo, Gingerbread, Honeycomb).
/// The Windows version is suitable for local testing
/// (Windows-to-Windows), the Android version can be executed
/// on a real Android device over the Android Debug Bridge
/// with activated TCP forwarding.
/// The Android device does NOT have to be MobiCore-enabled
/// because all cryptographic operations are simulated
/// by OpenSSL.

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <fcntl.h>
#ifdef WIN32
#include <io.h>
#endif

#include <tcpipnetworking.h>
#include <gdmcprovlib.h>

#ifdef WIN32
#pragma warning ( disable : 4996 )
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
static HKEY regKey = 0;
#endif // WIN32
#ifdef WIN32
#define AUTHTOK_NAME "00000000.authtokcont"
#define open  _open
#else
#define AUTHTOK_NAME "/data/app/mcRegistry/00000000.authtokcont"
#endif

static unsigned char msgin[4096];   ///< message input buffer

static unsigned char msgout[4096];  ///< message output buffer

#define SO_AUTHTOK_SIZE         152

/// a sample implementation of the WriteAuthTokenProc callback
static gderror WriteAuthTokenProc ( const _u8 *authtok, _u32 authtok_size )
{
  int fh;
#ifdef _DEBUG
  fprintf(stdout,"WriteAuthTokenProc called with size=%u.\n",authtok_size);
#endif // _DEBUG

  if (SO_AUTHTOK_SIZE!=authtok_size)
    return GDERROR_CANT_DUMP_AUTHTOKEN;

#ifdef WIN32
  fh = open(AUTHTOK_NAME,_O_BINARY|_O_CREAT|_O_WRONLY,0666);
```

```
#else
  fh = open(AUTHTOK_NAME,O_CREAT|O_WRONLY,0666);
#endif

  if (-1==fh)
    return GDERROR_CANT_DUMP_AUTHTOKEN;

  if (SO_AUTHTOK_SIZE!=write(fh,authtok,SO_AUTHTOK_SIZE))
  {
    close(fh);
    return GDERROR_CANT_DUMP_AUTHTOKEN;
  }

  close(fh);

  return GDERROR_OK;
}

/// a sample implementation of the ReadAuthTokenProc
static gderror ReadAuthTokenProc ( _u8 *authtok, _u32 *authtok_size )
{
  int fh;
#ifdef _DEBUG
  fprintf(stdout,"ReadAuthTokenProc called.\n");
#endif // _DEBUG

#ifdef WIN32
  fh = open(AUTHTOK_NAME,_O_RDONLY);
#else
  fh = open(AUTHTOK_NAME,O_RDONLY);
#endif

  if (-1==fh)
    return GDERROR_CANT_BUILD_AUTHTOKEN;

  if (*authtok_size<SO_AUTHTOK_SIZE)
    return GDERROR_CANT_BUILD_AUTHTOKEN;

  if (SO_AUTHTOK_SIZE!=read(fh,authtok,SO_AUTHTOK_SIZE))
  {
    close(fh);
    return GDERROR_CANT_BUILD_AUTHTOKEN;
  }

  close(fh);

  *authtok_size = SO_AUTHTOK_SIZE;

  return GDERROR_OK;
}

#define MAX_ERROR_COUNT       20

#ifdef WIN32

typedef void (*FuncmcRegistrySetTlsKey) ( DWORD key );

static HMODULE                  g_hMcRegistry;
static FuncmcRegistrySetTlsKey  g_mcRegistrySetTlsKey;
static DWORD                    g_tlsKey;

static void initialize_win32_specific ( void )
{
#ifdef _DEBUG
```

```
  g_hMcRegistry = LoadLibraryA("mcregistry32d.dll");
#else
  g_hMcRegistry = LoadLibraryA("mcregistry32.dll");
#endif

  if (NULL==g_hMcRegistry)
    return;

  g_mcRegistrySetTlsKey                                                        =
(FuncmcRegistrySetTlsKey)GetProcAddress(g_hMcRegistry,"mcRegistrySetTlsKey");

  if (NULL!=g_mcRegistrySetTlsKey)
  {
    g_tlsKey = TlsAlloc();
    g_mcRegistrySetTlsKey(g_tlsKey);
    TlsSetValue(g_tlsKey,(LPVOID)"."); // current path
  }
}

static void cleanup_win32_specific ( void )
{
  if (0!=g_tlsKey)
    TlsFree(g_tlsKey);
  FreeLibrary(g_hMcRegistry);
}

#endif // WIN32

int main ( int argc, char *argv[] )
{
  gderror             error;
  gdhandle            provhandle;
  _u32                msgin_size, msgout_size;
  int                 error_count = 0;

#ifdef WIN32
  initialize_win32_specific();
#endif

  // 0.) Override IP and port if specified (of communication partner)

  if (argc>1)
    comm_setip_port(argv[1]);

  // 1.) Initialize G&D Provisioning Library

  fprintf(stdout,"Initializing GDMCProvLibrary...");
  fflush(stdout);

  error=GDMCProvInitializeLibrary();
  if (GDERROR_OK!=error)
  {
    fprintf(stderr,"ERROR.\n");
    return 1;
  }
  fprintf(stdout,"OK.\n");

  // 2.) Set read/write SO.AuthToken callbacks

  GDMCProvSetAuthTokenCallbacks(WriteAuthTokenProc,ReadAuthTokenProc);

  // 3.) Acquire provisioning handle

  fprintf(stdout,"Acquiring provisioning handle...");
```

```
  fflush(stdout);

  error=GDMCProvBeginProvisioning(&provhandle);
  if (GDERROR_OK!=error)
  {
    GDMCProvShutdownLibrary();
    fprintf(stderr,"ERROR.\n");
    return 1;
  }
  fprintf(stdout,"OK.\n");

  // 4.) Initialize TCP/IP networking...

  fprintf(stdout,"Starting TCP/IP server and waiting for incoming data...");
  fflush(stdout);

  if (!comm_setup(1)) // we (Android side) are the server
  {
    fprintf(stderr,"ERROR.\n");
    return 1;
  }

  fprintf(stdout,"OK - continuing with provisioning loop.\n");

  // 5.) Perform the provisioning loop

  msgin_size = sizeof(msgin);

  while (comm_recv(msgin,&msgin_size))
  {
    msgout_size = sizeof(msgout);

#ifdef _DEBUG
    fprintf(stdout,"\nReceived %u byte(s), calling
GDMCProvExecuteProvisioningStep:\n",(unsigned int)msgin_size);
    gdmc_hexdump(msgin,msgin_size);
#endif

    error = GDMCProvExecuteProvisioningStep(
              provhandle,
              msgin,msgin_size,
              msgout,&msgout_size);

    if (0!=msgout_size)
    {
#ifdef _DEBUG
      fprintf(stdout,"Next message to be sent back contains %u
byte(s):\n",msgout_size);
      gdmc_hexdump(msgout,msgout_size);
      fprintf(stdout,"Trying to send...");
      fflush(stdout);
#endif // _DEBUG

      if (!comm_send(msgout,msgout_size))
      {
        fprintf(stderr,"ERROR (send).\n");
        comm_cleanup();
        GDMCProvEndProvisioning(provhandle);
        GDMCProvShutdownLibrary();
        return 1;
      }

#ifdef _DEBUG
      fprintf(stdout,"OK.\n");
```

```
#endif
    }

    if (GDERROR_PROVISIONING_DONE==error)
      break;

    if (GDERROR_OK!=error)
    {
      error_count++;
      if (MAX_ERROR_COUNT==error_count)
        fprintf(stderr,"ABOUT: maximum number of errors occurred. Giving up.\n");
      else
        fprintf(stderr,"ERROR: Provisioning error code %u occurred - trying to recover
by continuing the loop.\n",error);
    }

    msgin_size = sizeof(msgin); // set msgin_size for the next iteration
  } // of while

  comm_cleanup();
  GDMCProvEndProvisioning(provhandle);
  GDMCProvShutdownLibrary();

#ifdef WIN32
  cleanup_win32_specific();
#endif

  return 0;
}
```

B WSDL (SOAP B2B interface)

This appendix shows the ***W**eb* ***S**ervice* ***D**efinition* ***L**anguage* (WSDL) template file defining the SOAP B2B interface for sending receipt acknowledgement back to the OEM.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:ars="http://[OEM]/wsdl/AcknowledgeReceiptionService-v1"
        targetNamespace="http://[OEM]/wsdl/AcknowledgeReceiptionService-v1"
        name="AcknowledgeReceiptionService">

        <wsdl:types>
                <xsd:schema    targetNamespace="http://[OEM]/wsdl/AcknowledgeReceiptionService-
v1">

                        <xsd:simpleType name="ReceiptImportErrorCode">
                                <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="INSUFFICIENT_MEMORY" />
                                        <xsd:enumeration value="IVALID_TRAILING_CHARACTERS" />
                                        <xsd:enumeration value="INPUT_LINE_SYNTAX_ERROR" />
                                        <xsd:enumeration   value="INPUT_LINE_FIELD_NUMBER_INCORRECT"
/>
                                        <xsd:enumeration value="INPUT_FIELD_EXCEEDS_LIMIT" />
                                        <xsd:enumeration value="SDRECEIPT_BASE64_DECODE_ERROR" />
                                        <xsd:enumeration value="RSA_SIGNATURE_VALIDATION_FAILED" />
                                        <xsd:enumeration value="RSA_DECRYPTION_FAILED" />
                                        <xsd:enumeration value="SUID_MISMATCH" />
                                        <xsd:enumeration value="UNKNOWN_ERROR" />
                                        <xsd:enumeration value="BAD_REFURBISHFLAG" />
                                        <xsd:enumeration
value="UNABLE_TO_UPDATE_REFURBISHED_DEVICE" />
                                        <xsd:enumeration value="SET_USERDATA_FAILED" />
                                        <xsd:enumeration value="DEVICE_ALREADY_IN_DATABASE" />
                                        <xsd:enumeration value="INTERNAL_ERROR" />
                                </xsd:restriction>
                        </xsd:simpleType>

                        <xsd:simpleType name="ReceiptsImportStatusCode">
                                <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="SUCCESS" />
                                        <xsd:enumeration value="MD_ERROR" />
                                        <xsd:enumeration value="RECEIPTS_ERROR" />
                                </xsd:restriction>
                        </xsd:simpleType>

                        <xsd:simpleType name="AcknowledgeReceiptionStatus">
                                <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="SUCCESS" />
                                        <xsd:enumeration value="FAIL" />
                                </xsd:restriction>
                        </xsd:simpleType>

                        <xsd:complexType name="ReceiptErrorData">
                                <xsd:attribute name="suid" type="xsd:int" use="required"/>
                                <xsd:attribute     name="code"     type="ars:ReceiptImportErrorCode"
use="required"/>
                        </xsd:complexType>

                        <xsd:element name="ReceiptsAcknowledgeRequest">
                                <xsd:complexType>
                                        <xsd:sequence>
                                                <xsd:element      name="tid"      type="xsd:string"
nillable="false"/>
```

```
                                              <xsd:element name="processedReceipts" type="xsd:int"
nillable="false"/>
                                              <xsd:element            name="receiptsImportStatus"
type="ars:ReceiptsImportStatusCode" nillable="false"/>
                                              <xsd:element              name="erroneousReceipts"
type="ars:ReceiptErrorData" nillable="true" maxOccurs="unbounded" />
                                      </xsd:sequence>
                              </xsd:complexType>
                      </xsd:element>

                      <xsd:element name="ReceiptsAcknowledgeResponse">
                              <xsd:complexType>
                                      <xsd:sequence>
                                        <xsd:element            name="AcknowledgeReceiptionStatus"
type="ars:AcknowledgeReceiptionStatus" minOccurs="0" nillable="false"/>
                                              <xsd:element      name="AcknowledgeReceiptionMessage"
minOccurs="0" nillable="false">
                                              <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="4000" />
              </xsd:restriction>
            </xsd:simpleType>
            </xsd:element>
                                      </xsd:sequence>
                              </xsd:complexType>
                      </xsd:element>

              </xsd:schema>
        </wsdl:types>

        <wsdl:message name="ReceiptsAcknowledgeRequest">
              <wsdl:part element="ars:ReceiptsAcknowledgeRequest"
                      name="request" />
        </wsdl:message>

        <wsdl:message name="ReceiptsAcknowledgeResponse">
              <wsdl:part element="ars:ReceiptsAcknowledgeResponse"
                      name="response" />
        </wsdl:message>

        <wsdl:portType name="AcknowledgeReceiptionService">
              <wsdl:operation name="handleReceiptsAcknowledge">
                      <wsdl:input message="ars:ReceiptsAcknowledgeRequest" />
                      <wsdl:output message="ars:ReceiptsAcknowledgeResponse" />
              </wsdl:operation>
        </wsdl:portType>

        <wsdl:binding name="AcknowledgeReceiptionServicePort"
              type="ars:AcknowledgeReceiptionService">
              <soap:binding style="document"
                      transport="http://schemas.xmlsoap.org/soap/http" />

              <wsdl:operation name="handleReceiptsAcknowledge">
                      <soap:operation soapAction="urn:any" style="document" />
                      <wsdl:input>
                              <soap:body use="literal" />
                      </wsdl:input>
                      <wsdl:output>
                              <soap:body use="literal" />
                      </wsdl:output>
              </wsdl:operation>
        </wsdl:binding>

        <wsdl:service name="AcknowledgeReceiptionService">
              <wsdl:port binding="ars:AcknowledgeReceiptionServicePort"
                      name="AcknowledgeReceiptionServicePort">
                      <soap:address
                              location="https://[OEM-WebAddress]:[OEM-
Port]/AcknowledgeReceiptionService/" />
              </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```